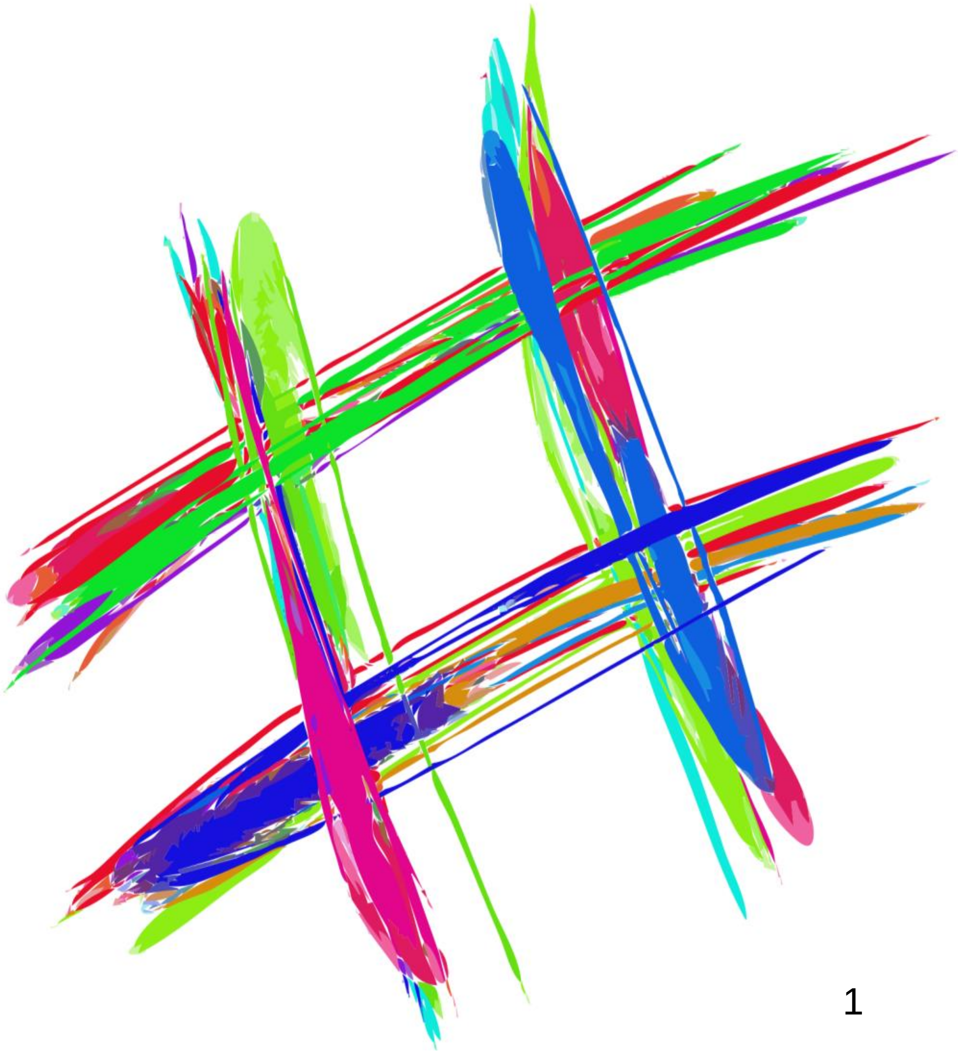
Lecture 7: Efficient Collections via Hashing

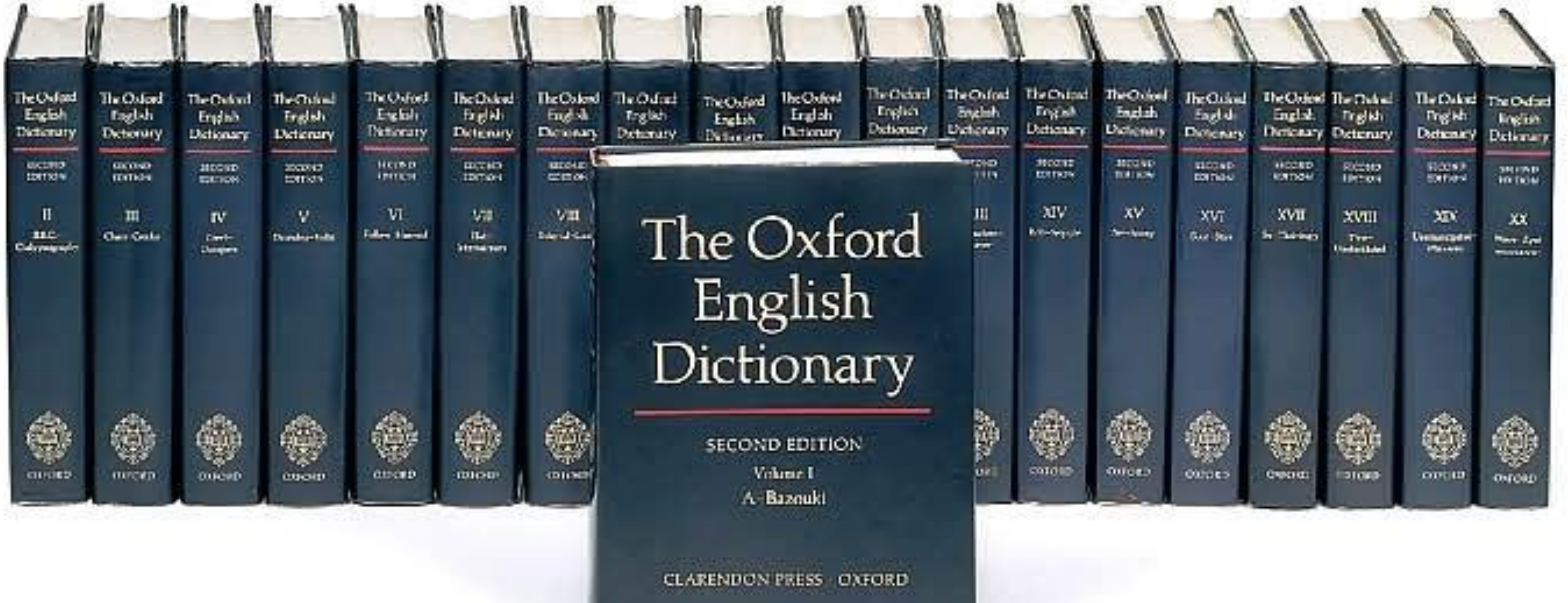


These slides include material originally prepared by Dr. Ron Cytron, Dr. Jeremy Buhler, and Dr. Steve Cole.

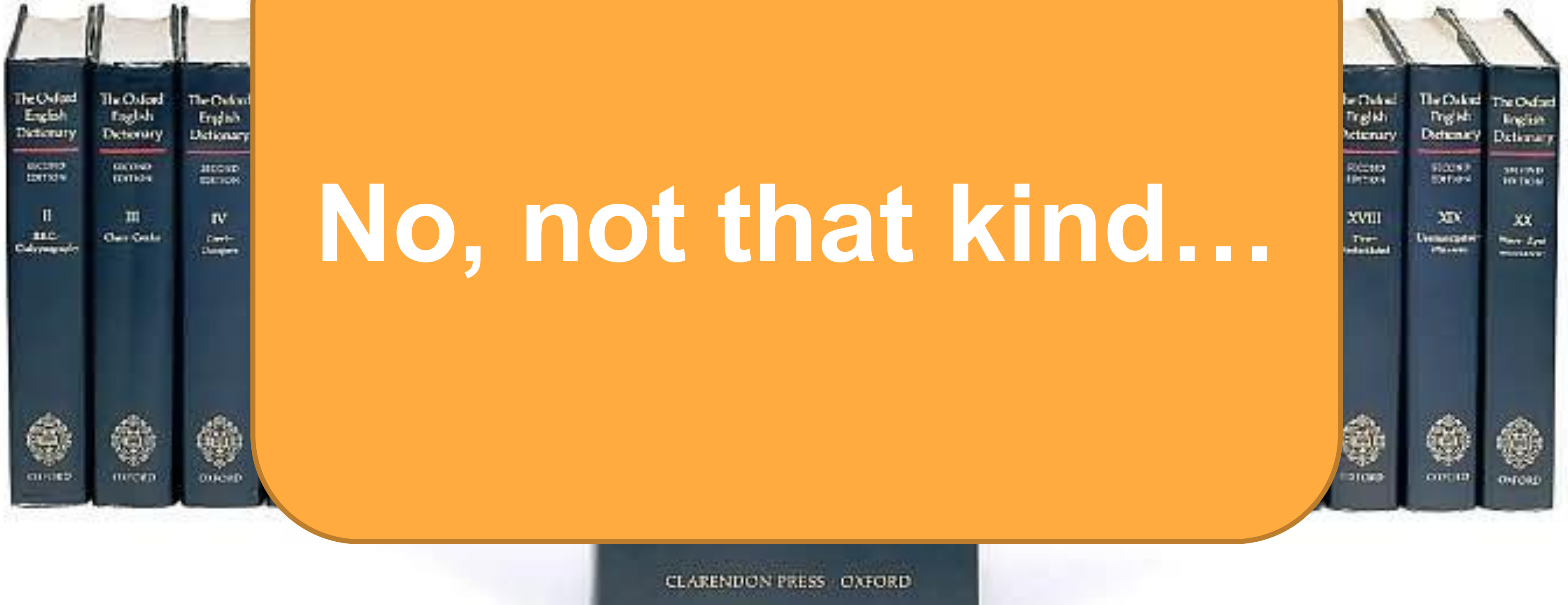
Announcements

- **Lab 6** due Friday
- **Lab 7** out tomorrow – all about hashing!
- Pre-lab due 3/19; code and post-lab due 3/22
- Spring Break hours
  - No official course communication from 3/8 evening (Friday) until 3/18 morning (Monday)
    - Please be patient on Piazza: instructor and TAs are on break too :-)

Let’s Talk About Dictionaries



Let’s Talk About Dictionaries



No, not that kind…

Dictionary ADT

- A **dictionary** is a data structure that stores a collection of objects
- Each object is associated with a **key**
- Objects can be dynamically inserted and removed
- Can efficiently find an object in the dictionary by its key

Dictionary Operations (One of Several Versions)

- **insert(Record r)** – add r to the dictionary
- **find(Key k)** – return one/some/all records whose key matches k, if any
- **remove(Key k)** – remove all records whose key matches k, if any

Dictionary Operations (One of Several Versions)

- **insert(Record r)** – add r to the dictionary
- **find(Key k)** – return one/some/all records whose key matches k, if any
- **remove(Key k)** – remove all records whose key matches k, if any
- *Other versions are possible, e.g. remove() might take a Record to remove, rather than a key*

Dictionary Operations (One of Several Versions)

- **insert(Record r)** – add r to the dictionary
- **find(Key k)** – return one/some/all records whose key matches k, if any
- **remove(Key k)** – remove all records whose key matches k, if any
- *Other ops may exist, e.g. isEmpty(), size(),* ***iterator()***

Dictionary Examples

- An actual dictionary – a collection that maps words to definitions
- Class list – a set of students, with name (or possibly ID) as key
- DMV database – a collection of cars accessed by license plate number
- …

Some Questions about Dictionary Variants

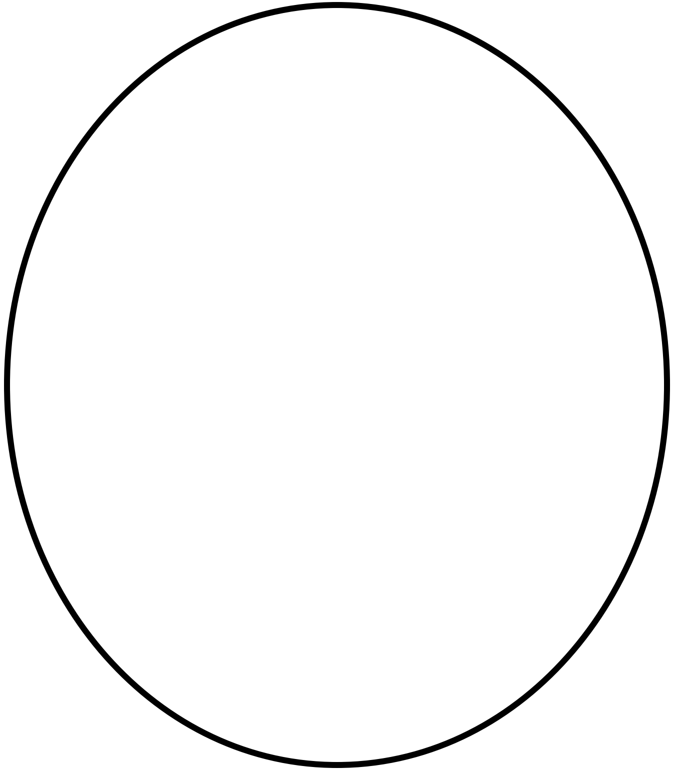
- Can multiple records exist in dictionary with same key?

- What happens if find() does not find a record with a specified key?

- Is key the entire record (as in Java **Set** interface), is it internal to the record (as in Lab 7), or is it external (as in Java **Map** interface)?

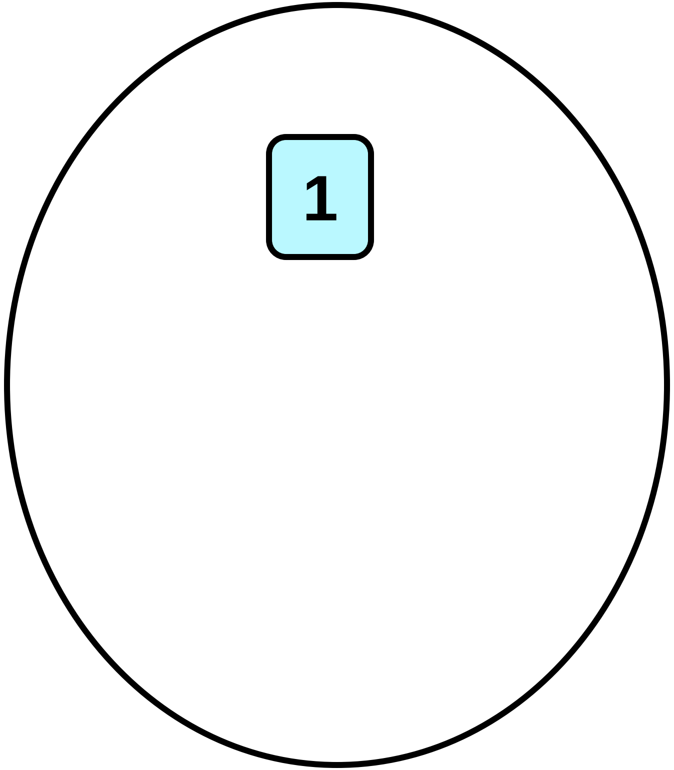
How to Build a Dictionary

- Conceptually, it’s just “bag of records”
- What concrete data structure do we use to implement it?
- Must support efficient dynamic add/remove *and* find



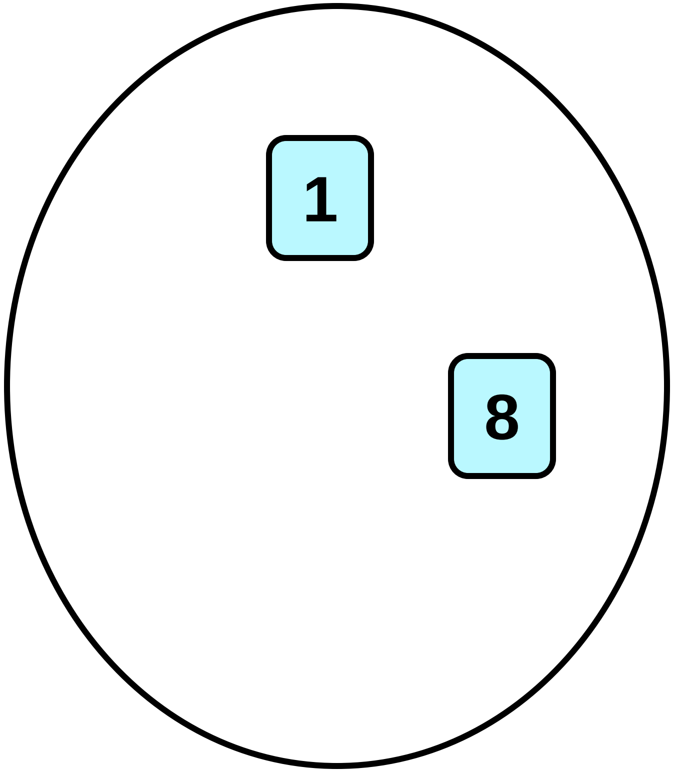
How to Build a Dictionary

- Conceptually, it's just "bag of records"
- What concrete data structure do we use to implement it?
- Must support efficient dynamic add/remove *and* find
- **insert**



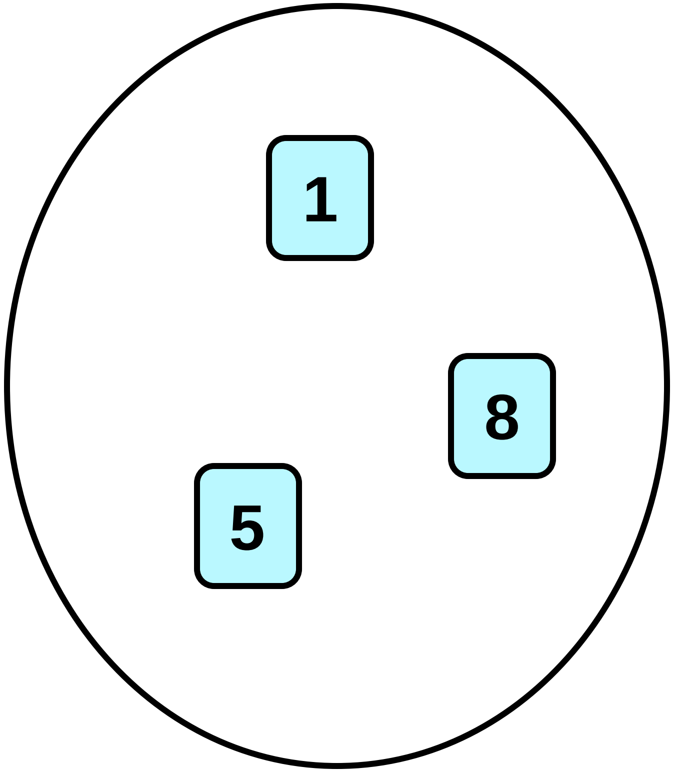
How to Build a Dictionary

- Conceptually, it's just "bag of records"
- What concrete data structure do we use to implement it?
- Must support efficient dynamic add/remove *and* find
- **insert**



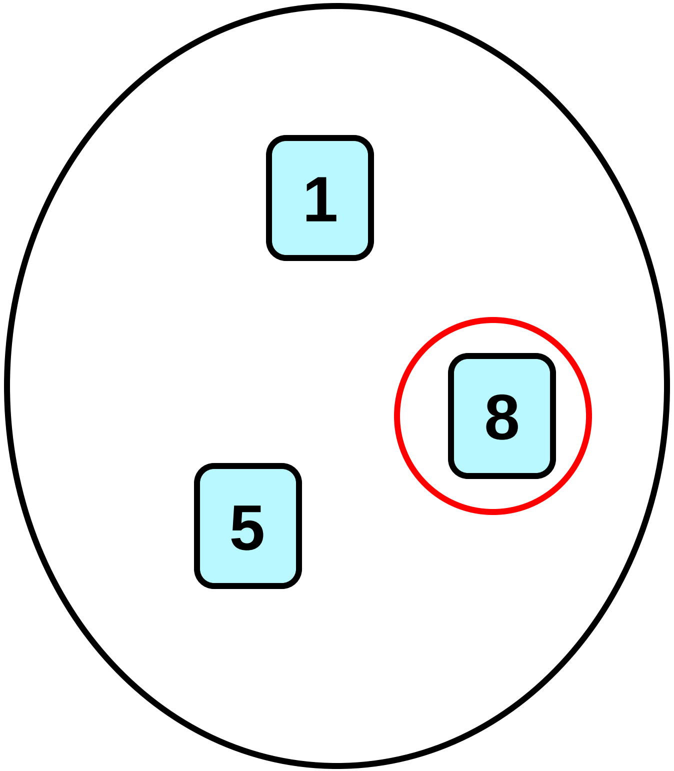
How to Build a Dictionary

- Conceptually, it's just "bag of records"
- What concrete data structure do we use to implement it?
- Must support efficient dynamic add/remove *and* find
- **insert**



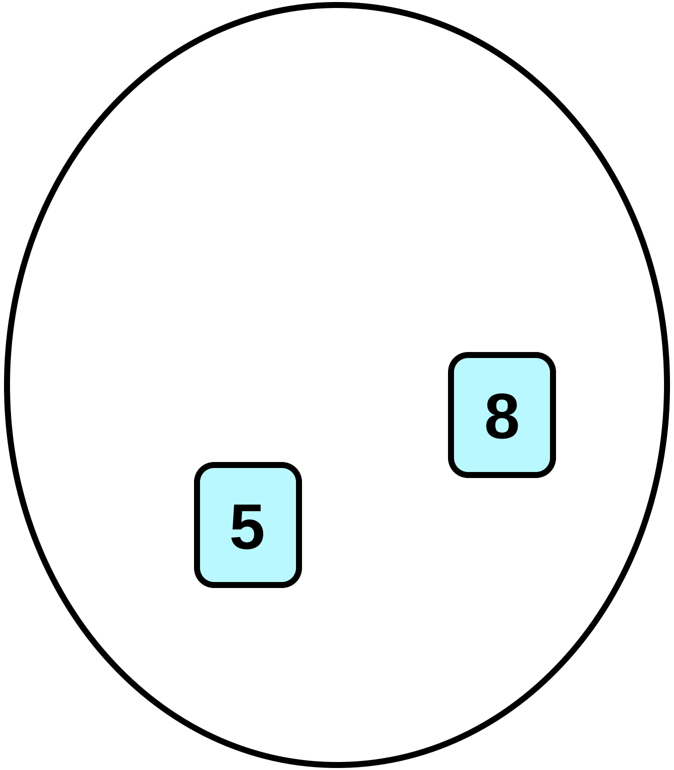
How to Build a Dictionary

- Conceptually, it's just "bag of records"
- What concrete data structure do we use to implement it?
- Must support efficient dynamic add/remove *and* find
- **find(8)**



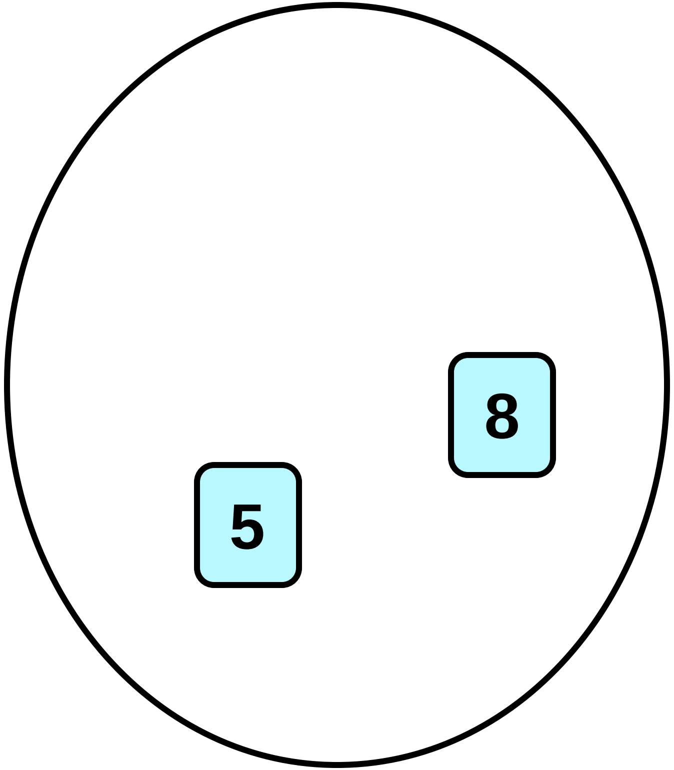
How to Build a Dictionary

- Conceptually, it's just "bag of records"
- What concrete data structure do we use to implement it?
- Must support efficient dynamic add/remove *and* find
- **delete(1)**



How to Build a Dictionary

- Conceptually, it’s just “bag of records”
- What concrete data structure do we use to implement it?
- Must support efficient dynamic add/remove *and* find
- **find(1) → “not found”**



Some bad implementations

- Time complexities for dictionary operations

Structure	`insert`	`delete`	`find`	`space`
unsorted list	Θ(?)	Θ(?)	Θ(?)	Θ(?)
sorted list	Θ(?)	Θ(?)	Θ(?)	Θ(?)
sorted array	Θ(?)	Θ(?)	Θ(?)	Θ(?)
min-heap	Θ(?)	Θ(?)	Θ(?)	Θ(?)

Some bad implementations

- Time complexities for dictionary operations

Structure	**`insert`**	**`delete`**	**`find`**	**`space`**
unsorted list	**$\Theta(1)$**	**$\Theta(n)$**	**$\Theta(n)$**	$\Theta(?)$
sorted list	$\Theta(?)$	$\Theta(?)$	$\Theta(?)$	$\Theta(?)$
sorted array	$\Theta(?)$	$\Theta(?)$	$\Theta(?)$	$\Theta(?)$
min-heap	$\Theta(?)$	$\Theta(?)$	$\Theta(?)$	$\Theta(?)$

Some bad implementations

- Time complexities for dictionary operations

Structure	`insert`	`delete`	`find`	`space`
unsorted list	**Θ(1)**	**Θ(n)**	**Θ(n)**	Θ(?)
sorted list	Θ(?)	Θ(?)	Θ(?)	Θ(?)
sorted array	Θ(?)	Θ(?)	Θ(?)	Θ(?)
min-heap	Θ(?)	Θ(?)	Θ(?)	Θ(?)

- What assumption is being made about `delete` ? Any other assumptions here?

Some bad implementations

- Time complexities for dictionary operations

Structure	`insert`	`delete`	`find`	`space`
unsorted list	Θ(1)	Θ(n)	Θ(n)	Θ(?)
sorted list	**Θ(n)**	**Θ(n)**	**Θ(n)**	Θ(?)
sorted array	Θ(?)	Θ(?)	Θ(?)	Θ(?)
min-heap	Θ(?)	Θ(?)	Θ(?)	Θ(?)

Some bad implementations

- Time complexities for dictionary operations

Structure	`insert`	`delete`	`find`	`space`
unsorted list	Θ(1)	Θ(n)	Θ(n)	Θ(?)
sorted list	Θ(n)	Θ(n)	Θ(n)	Θ(?)
sorted array	**Θ(n)**	**Θ(n)**	**Θ(log n)**	Θ(?)
min-heap	Θ(?)	Θ(?)	Θ(?)	Θ(?)

Some bad implementations

- Time complexities for dictionary operations

Structure	insert	delete	find	space
unsorted list	Θ(1)	Θ(n)	Θ(n)	Θ(?)
sorted list	Θ(n)	Θ(n)	Θ(n)	Θ(?)
sorted array	Θ(n)	Θ(n)	Θ(log n)	Θ(?)
min-heap	**Θ(log n)**	**XXX**	**XXX**	Θ(?)

Heaps don't support these ops
(but find would be Θ(n))

Some bad implementations

- Time complexities for dictionary operations

Structure	`insert`	`delete`	`find`	`space`
unsorted list	Θ(1)	Θ(n)	Θ(n)	Θ(?)
sorted list	Θ(n)	Θ(n)	Θ(n)	Θ(?)
sorted array	Θ(n)	Θ(n)	Θ(log n)	Θ(?)
min-heap	**Θ(log n)**	**XXX**	**XXX**	Θ(?)

None of these structures achieve sublinear time complexity for all three ops

Some bad implementations

- Time complexities for dictionary operations

Structure	`insert`	`delete`	`find`	space
unsorted list	Θ(1)	Θ(n)	Θ(n)	Θ(?)
sorted list	Θ(n)	Θ(n)	Θ(n)	Θ(?)
sorted array	Θ(n)	Θ(n)	Θ(log n)	Θ(?)
min-heap	**Θ(log n)**	**XXX**	**XXX**	Θ(?)

Some bad implementations

- Time complexities for dictionary operations

Structure	`insert`	`delete`	`find`	space
unsorted list	Θ(1)	Θ(n)	Θ(n)	Θ(n)
sorted list	Θ(n)	Θ(n)	Θ(n)	Θ(n)
sorted array	Θ(n)	Θ(n)	Θ(log n)	Θ(n)
min-heap	**Θ(log n)**	**XXX**	**XXX**	Θ(n)

Some bad implementations

- Time complexities for dictionary operations

Structure	`insert`	`delete`	`find`	space
unsorted list	Θ(1)	Θ(n)	Θ(n)	Θ(n)
sorted list	Θ(n)	Θ(n)	Θ(n)	Θ(n)
sorted array	Θ(n)	Θ(n)	Θ(log n)	Θ(n)
min-heap	**Θ(log n)**	**XXX**	**XXX**	Θ(n)

All these structures take space proportional to # of records stored

Key Question

- **Is it possible to implement a dictionary with sublinear time for all of insert, find, and remove?**

Key Question

- **Is it possible to implement a dictionary with sublinear time for all of insert, find, and remove?**

- We'll show that the answer is yes…

Key Question

- **Is it possible to implement a dictionary with sublinear time for all of insert, find, and remove?**

- We'll show that the answer is **yes**…

- …depending on what you mean by "sublinear time".

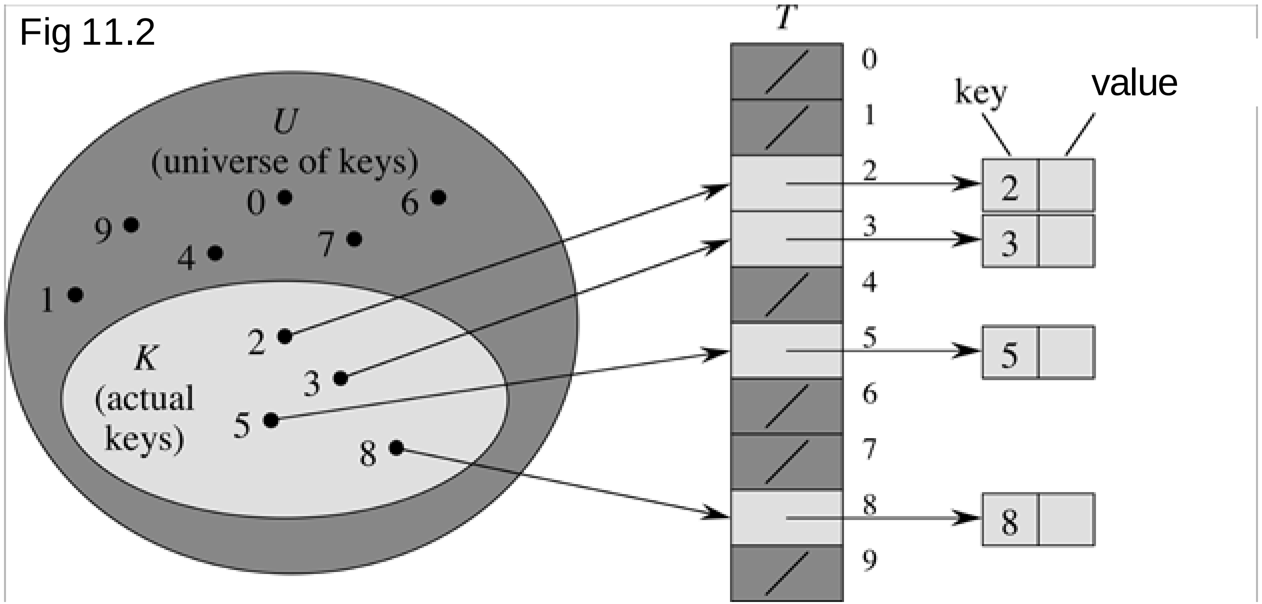
- (Guarantees will not be worst-case)

Idea: Direct-Addressed Table

- Let **U** be the set ("*universe*") of all *possible* keys
- Allocate an array of size **|U|**
- If we get a record with key k, put it in k's array cell.

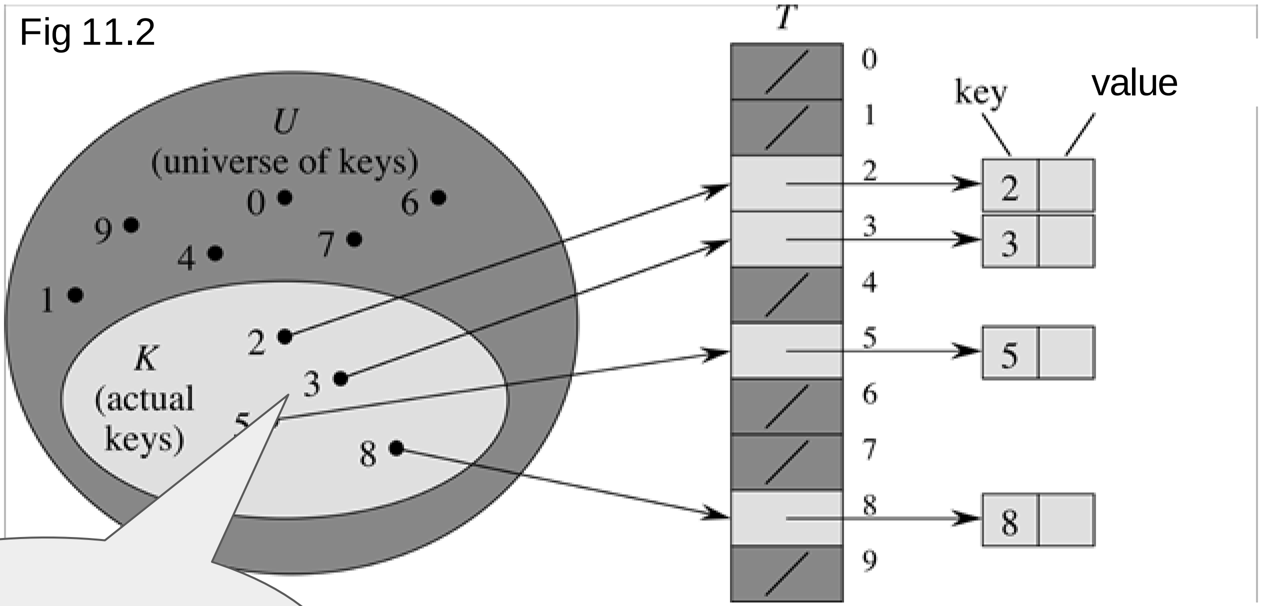
Direct-Addressed Tables

Fig 11.2



Direct-Addressed Tables

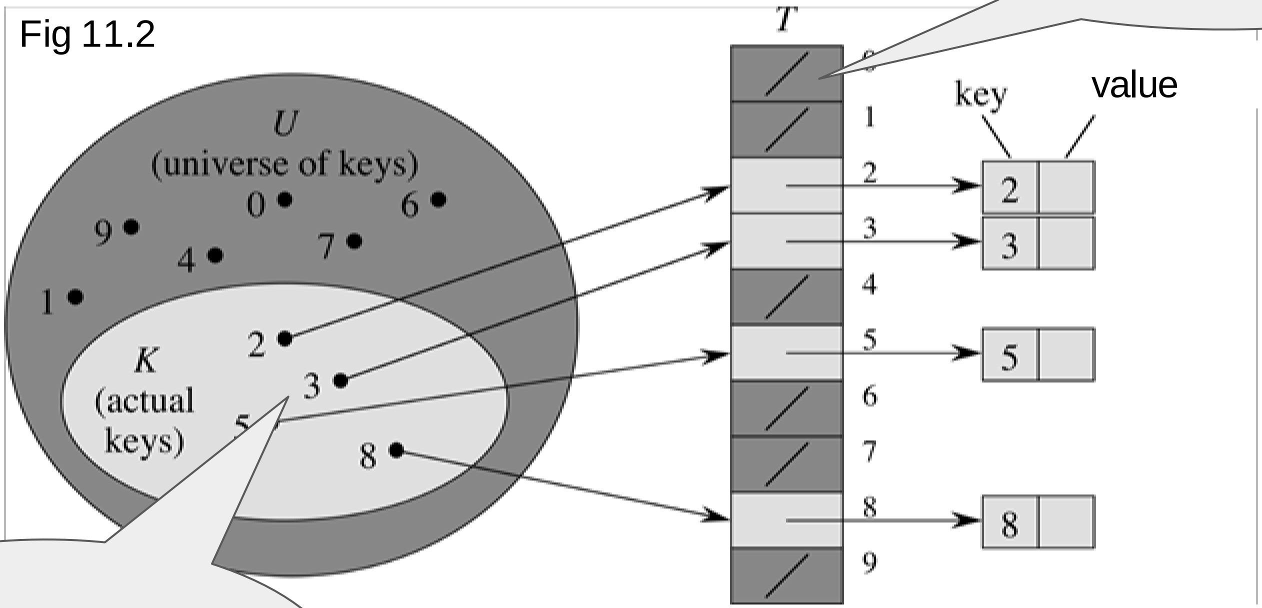
Fig 11.2



Key space is a compact index of small, nonnegative integers

Direct-Addressed Tables

Fig 11.2

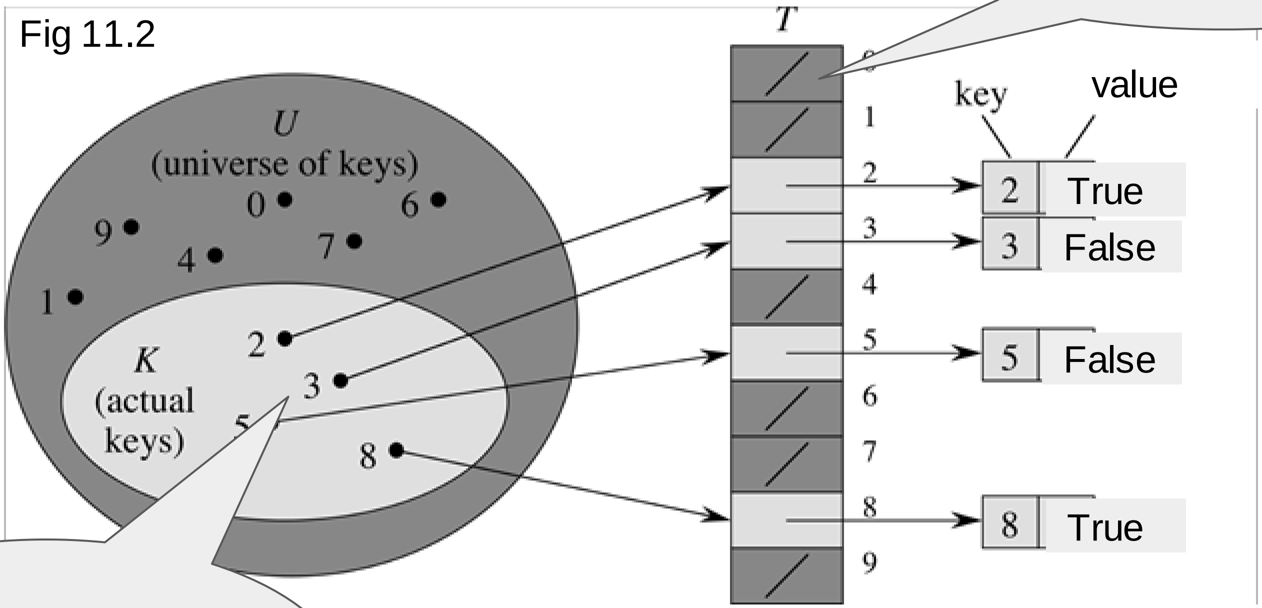


Darkened cells are all `null`

Key space is a compact index of small, nonnegative integers

Direct-Addressed Tables

Fig 11.2



Darkened cells are all `null`

Key space is a compact index of small, nonnegative integers

Example: record whether each key is divisible by 2

A Less Bad Implementation?

- Time complexities for dictionary operations

Structure	`insert`	`delete`	`find`	`space`
unsorted list	Θ(1)	Θ(n)	Θ(n)	Θ(n)
sorted list	Θ(n)	Θ(n)	Θ(n)	Θ(n)
sorted array	Θ(n)	Θ(n)	Θ(log n)	Θ(n)
direct table	Θ(???)	Θ(???)	Θ(???)	Θ(???)

A Less Bad Implementation?

- Time complexities for dictionary operations

Structure	`insert`	`delete`	`find`	`space`
unsorted list	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
sorted list	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
sorted array	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
direct table	**$\Theta(1)$**	**$\Theta(1)$**	**$\Theta(1)$**	$\Theta(???)$

We can look up any entry in the table in constant time, given its key

A Less Bad Implementation?

- Time complexities for dictionary operations

Structure	insert	delete	find	space
unsorted list	Θ(1)	Θ(n)	Θ(n)	Θ(n)
sorted list	Θ(n)	Θ(n)	Θ(n)	Θ(n)
sorted array	Θ(n)	Θ(n)	Θ(log n)	Θ(n)
direct table	**Θ(1)**	**Θ(1)**	**Θ(1)**	**Θ(|U|)**

But the space cost is |U|, no matter how small n (# of records) is.

Problems with Direct-Addressed Tables

- Challenge #1: What if |U| >> n?
  - ex. IPv6 (~10^{38}), Unix passwords (~10^{15})

Problems with Direct-Addressed Tables

- Challenge #1: What if |U| >> n?
  - ex. IPv6 (~10^{38}), Unix passwords (~10^{15})

- Challenge #2: What if keys aren't integers?
  - What does T[blue] mean? T[5.7281934]? T["hello world"]?
  - How do you index an array using an arbitrary object type?

Idea: Hash Functions

- A **hash function h** maps keys k of some type to integers h(k) in a fixed range *[0, N)*

- The integer h(k) is the key's **hashcode** under h

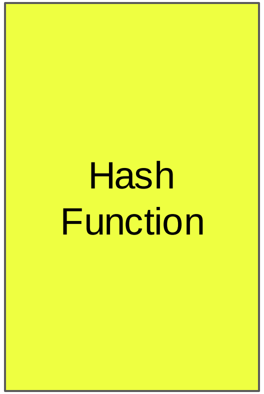
- If N = |U|, h could map every key to a *distinct* integer, giving us a way to index our direct table.

What if our key is not Integer?

"dog"

"cat"

"fossa"



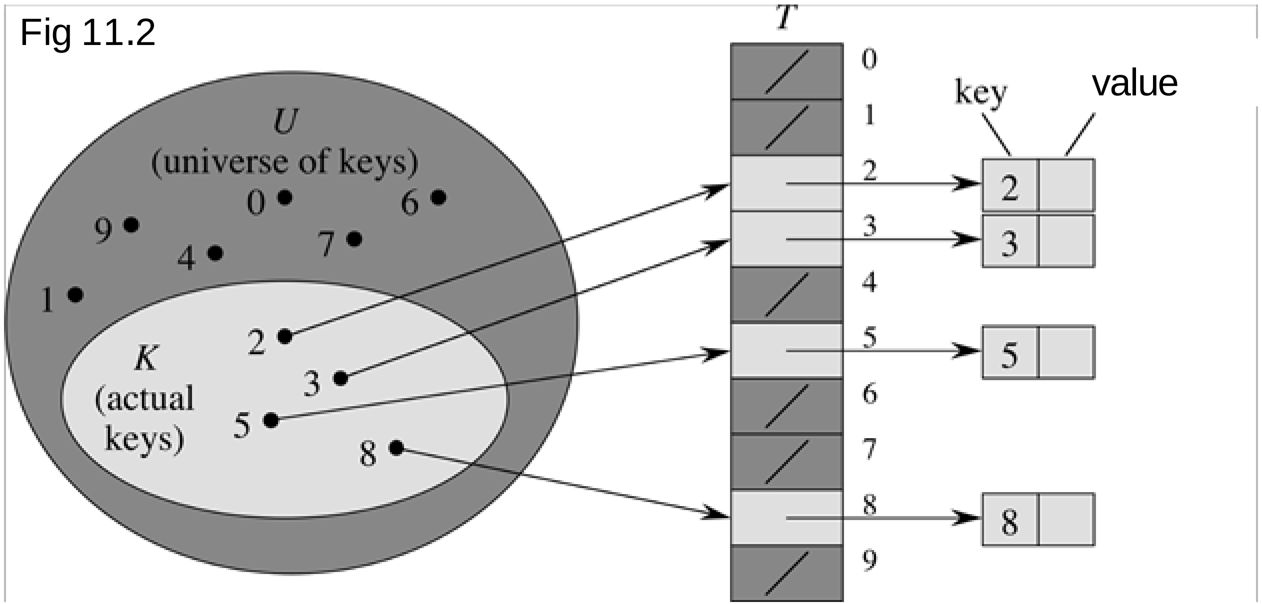


Fig 11.2

What if our key is not Integer?

“dog”

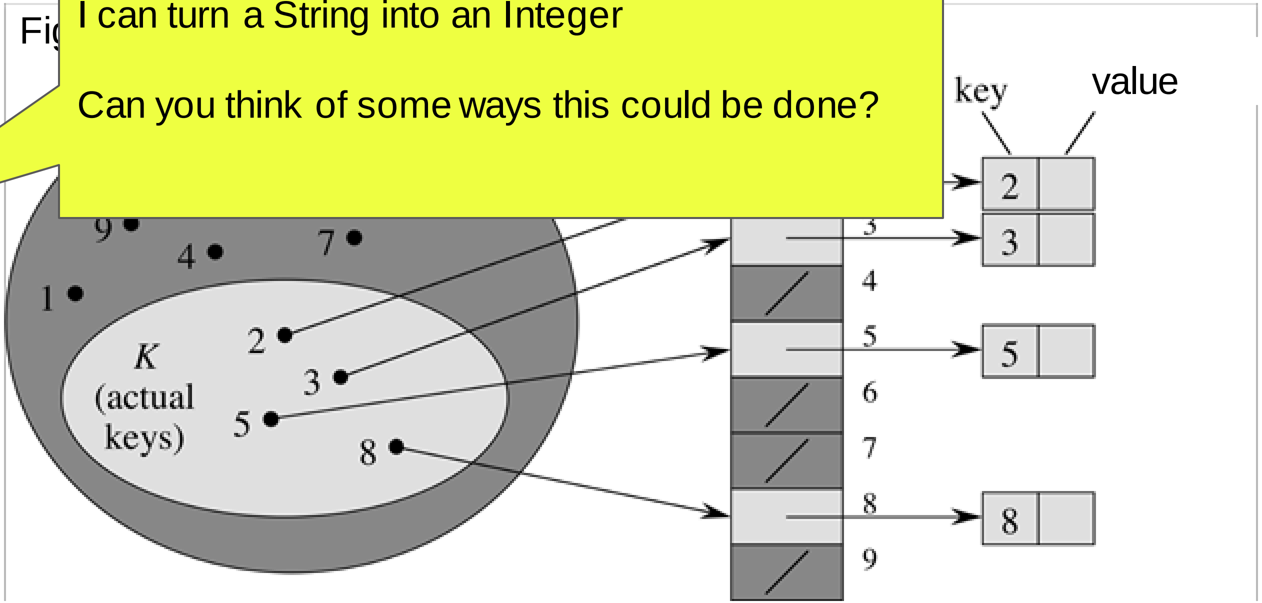
“cat”

“fossa”

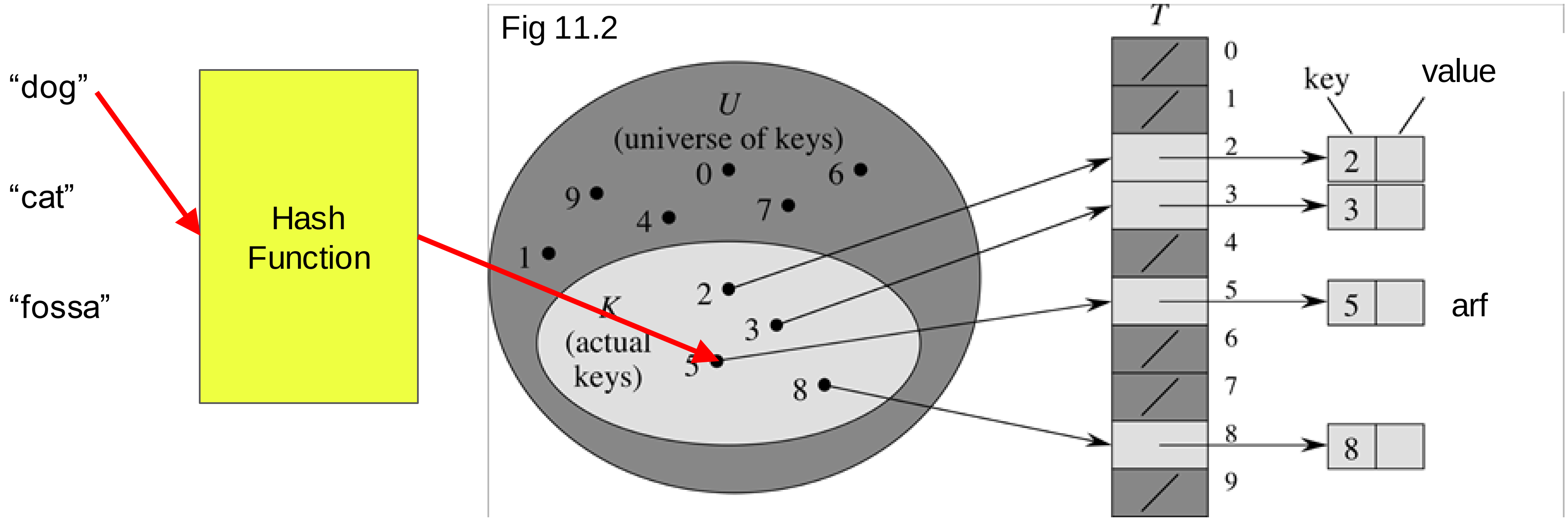
Hash Function

I can turn a String into an Integer

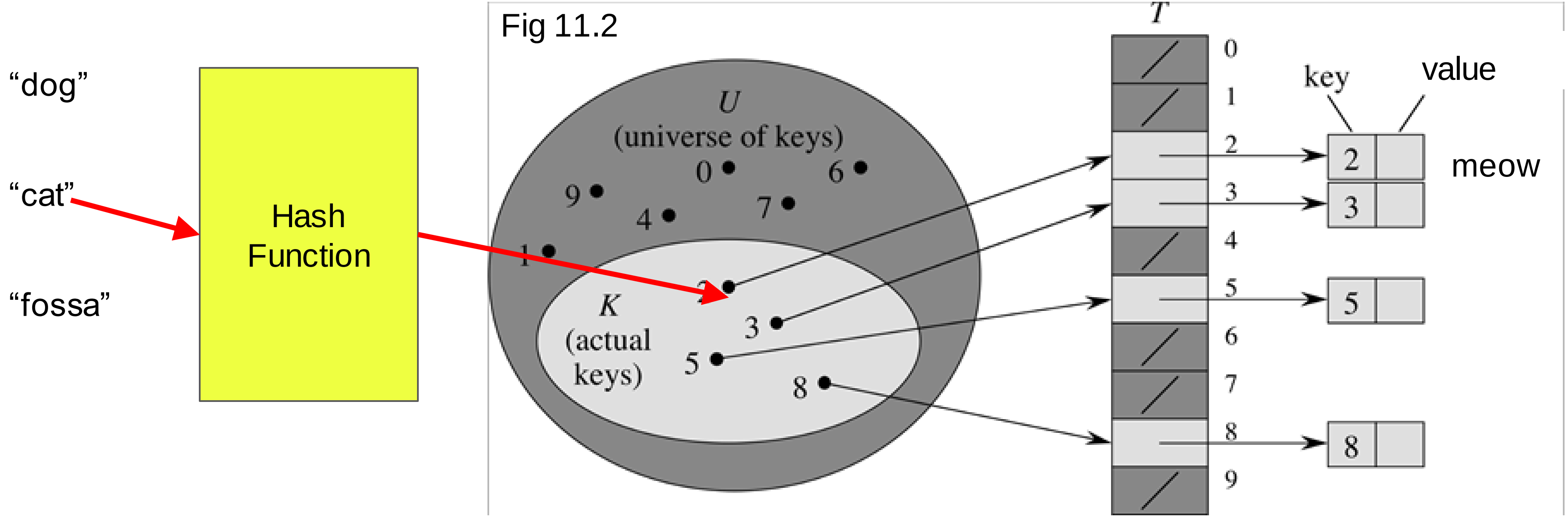
Can you think of some ways this could be done?



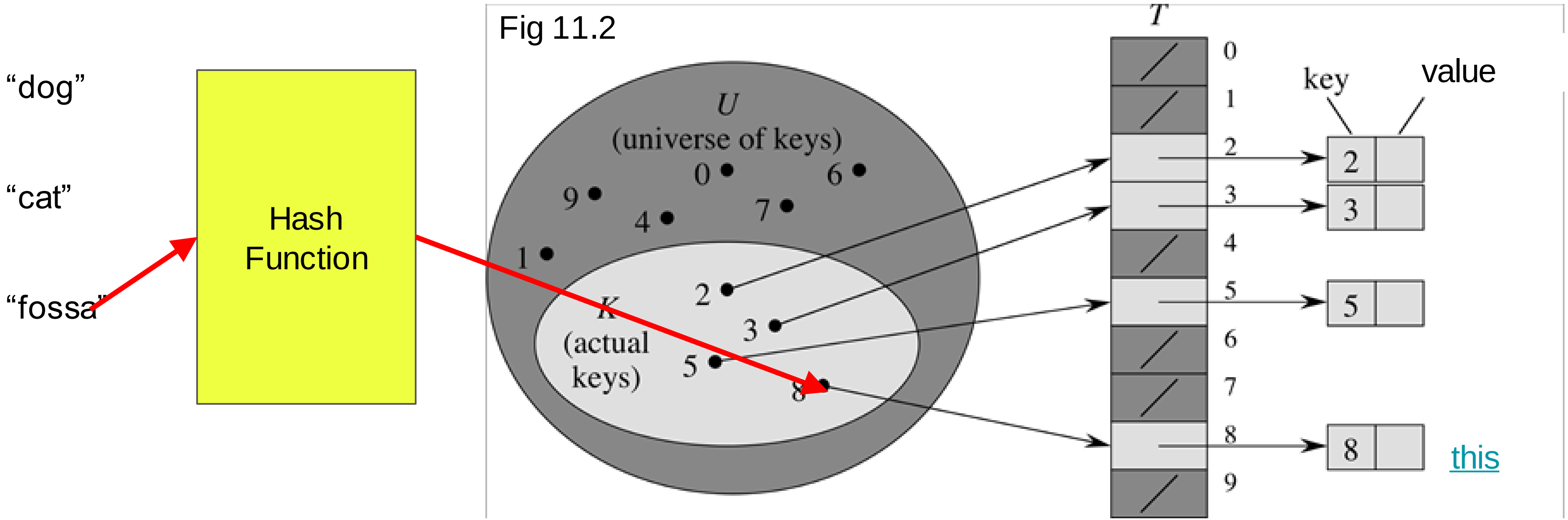
What if our key is not Integer?



What if our key is not Integer?



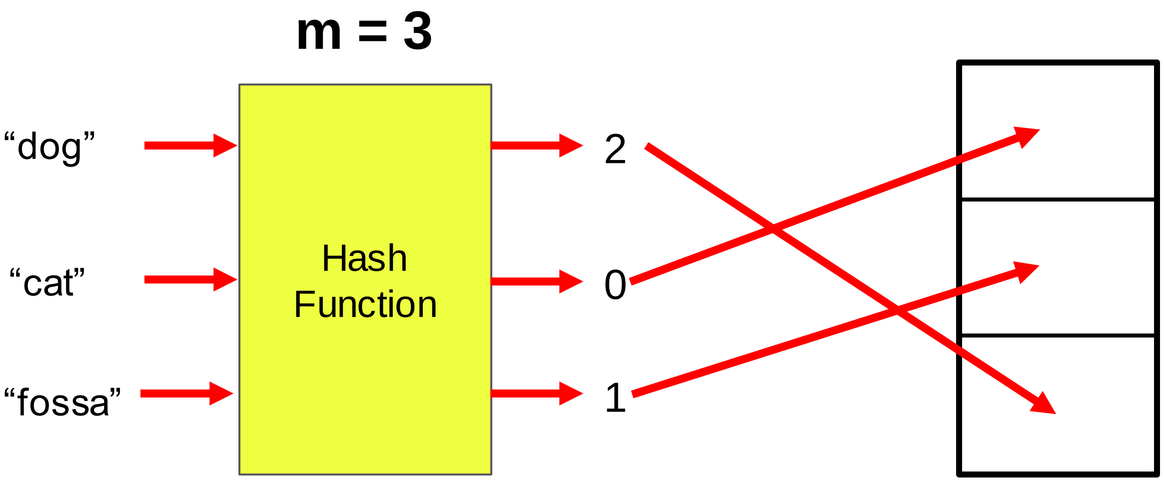
What if our key is not Integer?



But What About Sparsity?

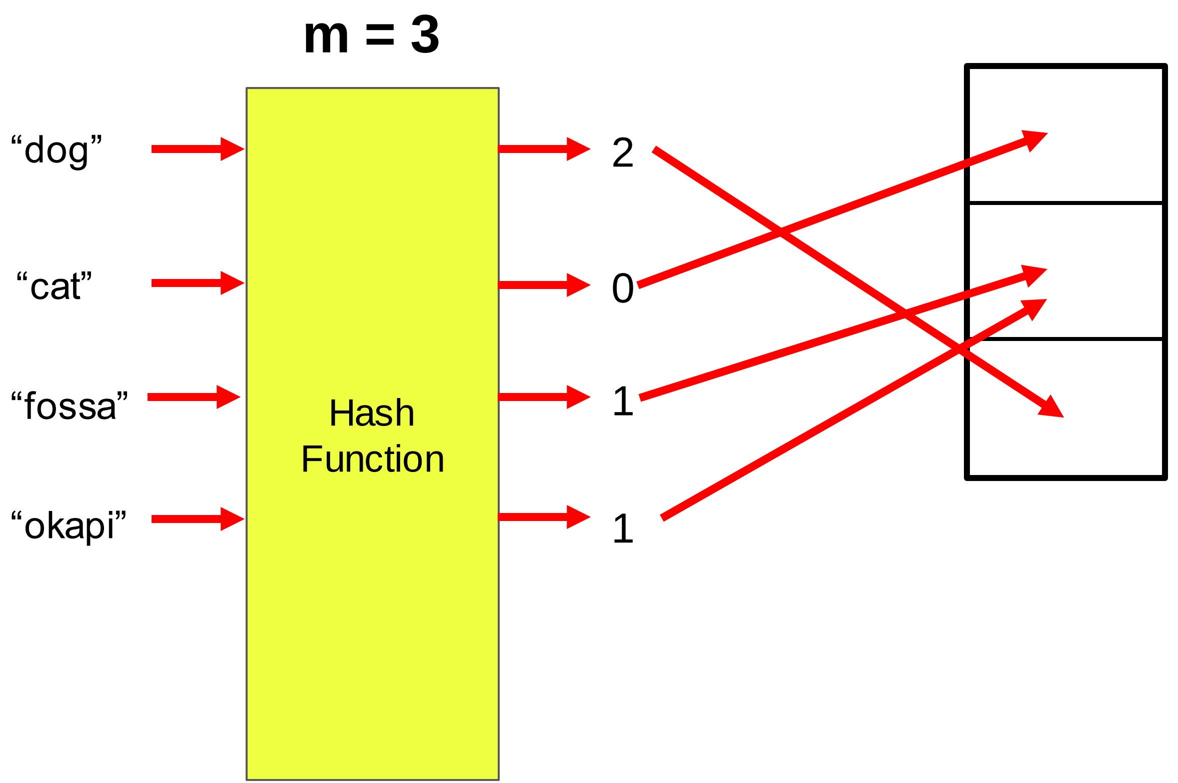
- We often can't afford to store a table of size |U|.
- What if our hash function mapped keys to a smaller space, i.e. [0, m) for m << |U|?
- We'd need a table of size only m.
- This smaller table is called a **hash table.**

The Good…



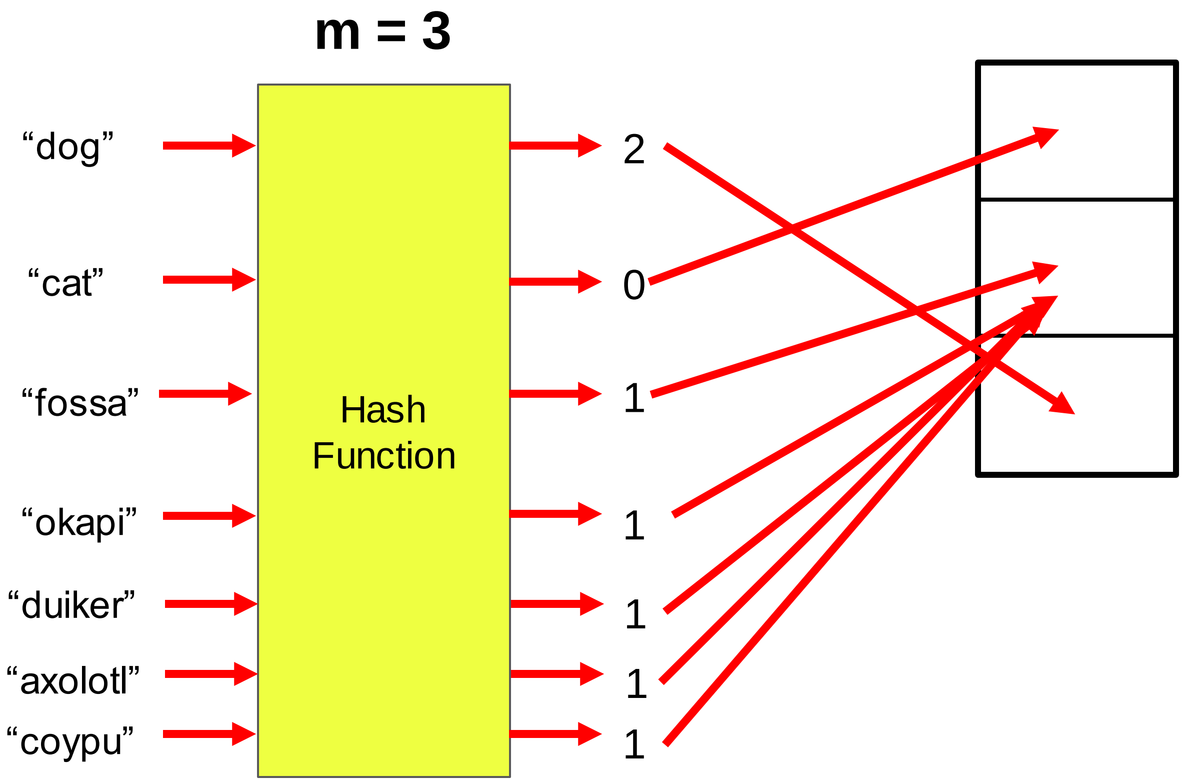
A hash table lets us allocate arrays much smaller than |U|

The Bad…



Uh oh…

The Bad…



Oh dear…

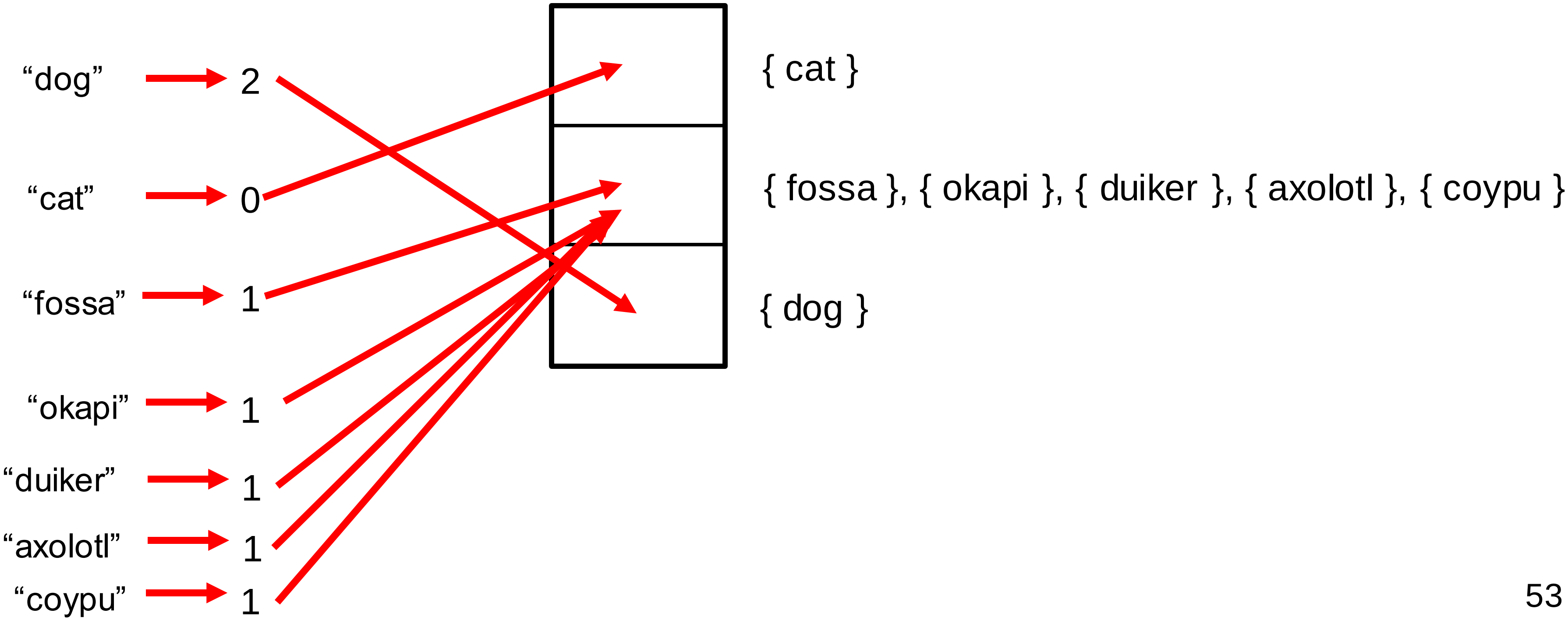
When ~~Worlds~~ Keys Collide

- What happens if multiple keys hash to same table cell?
- This **must** happen if m < |U| -- *pigeonhole principle*
- When two keys hash to same cell, we say they **collide**.
- *A hash table must work even in presence of collisions.*

A Simple Strategy: Chaining

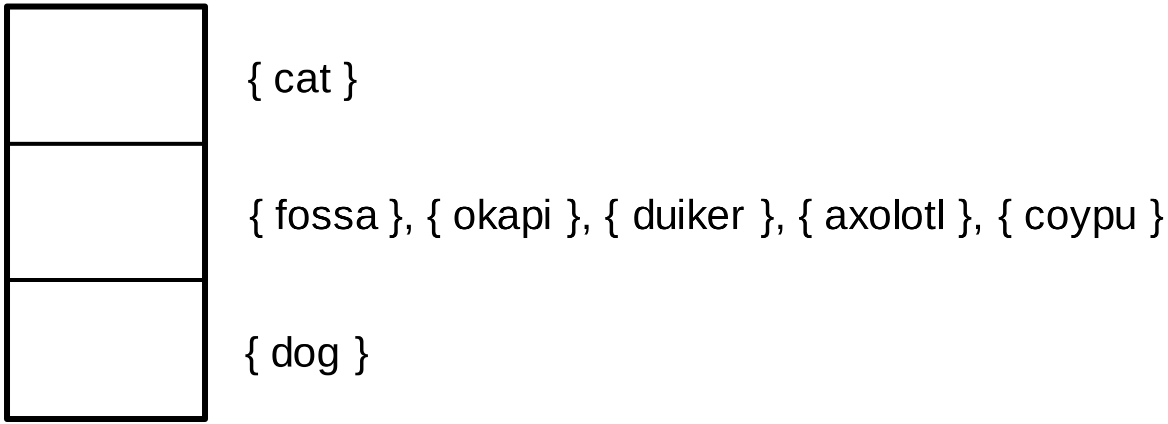
- Each table cell becomes a **bucket** that can hold multiple records
- A bucket holds a list of all records whose keys map to it.
- *find(k) must traverse bucket h(k)'s list*, looking for a record with key k
- Analogous extensions for insert(), remove()

Hash Table with Chaining

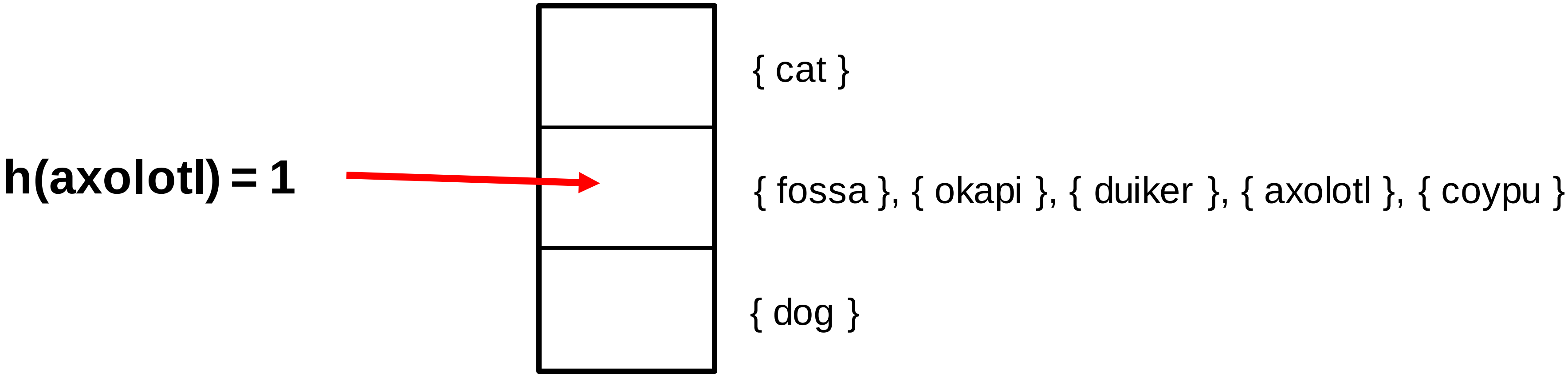


Hash Table with Chaining

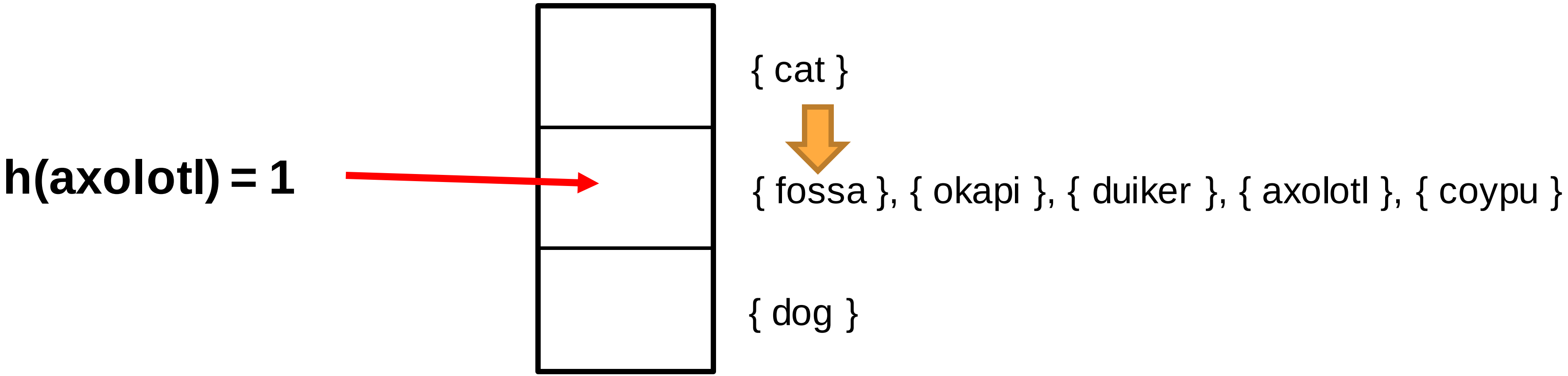
find(axolotl)



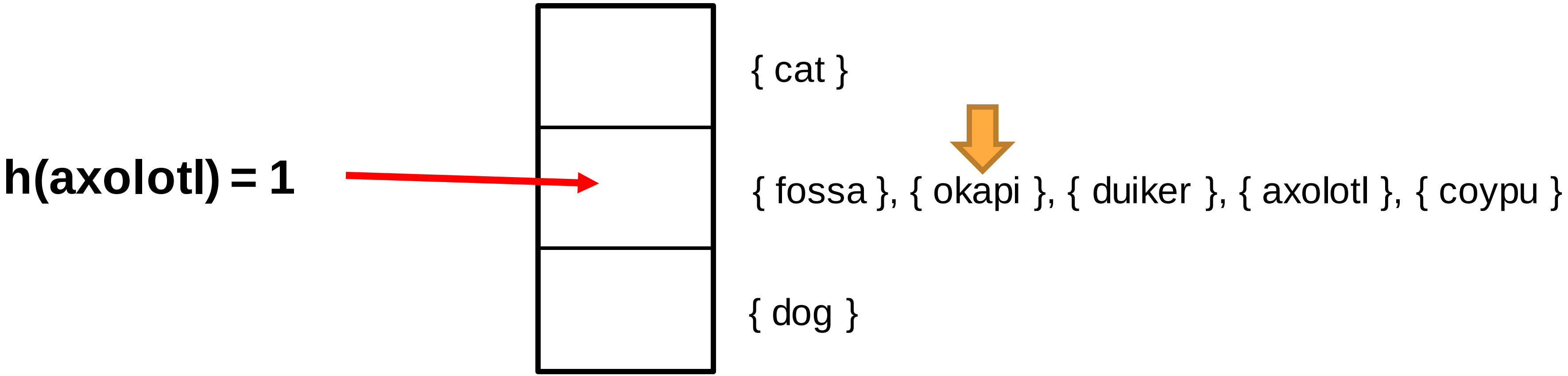
Hash Table with Chaining



Hash Table with Chaining

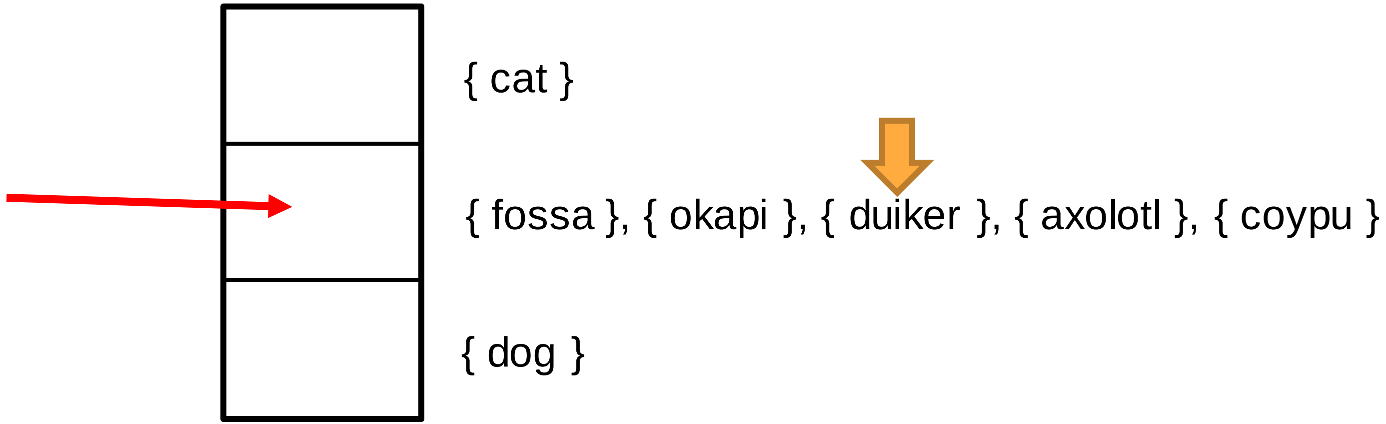


Hash Table with Chaining

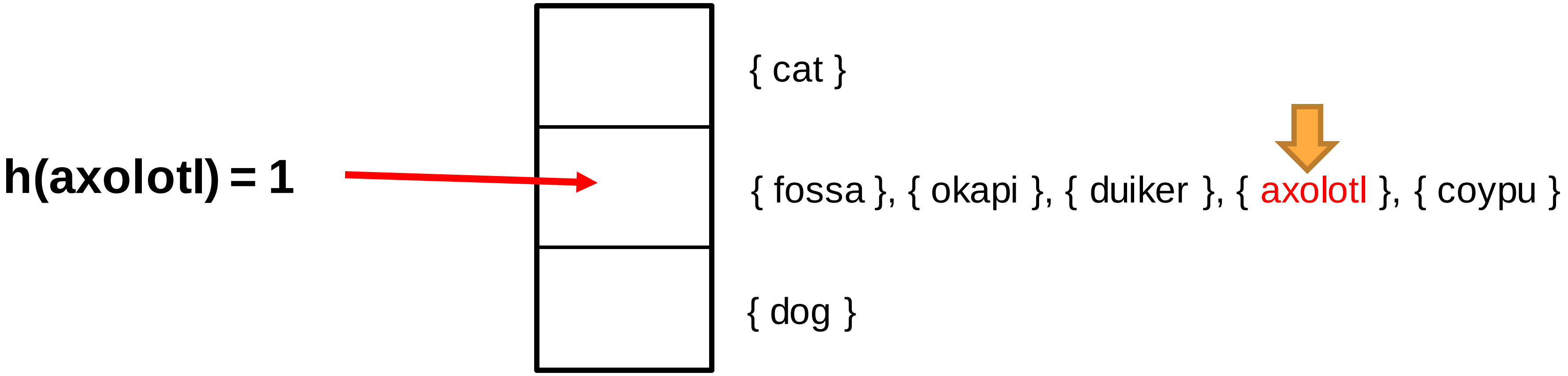


Hash Table with Chaining

h(axolotl) = 1

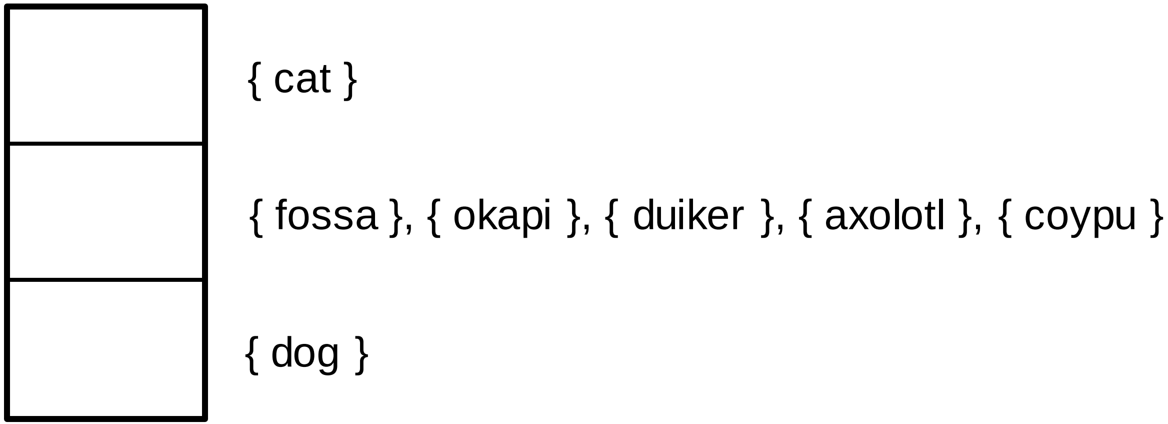


Hash Table with Chaining

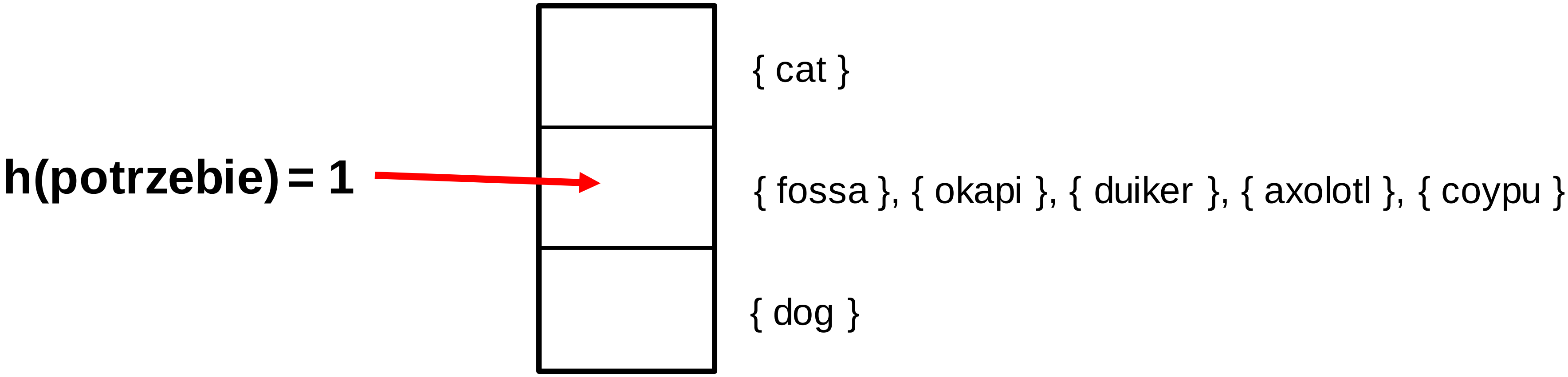


Hash Table with Chaining

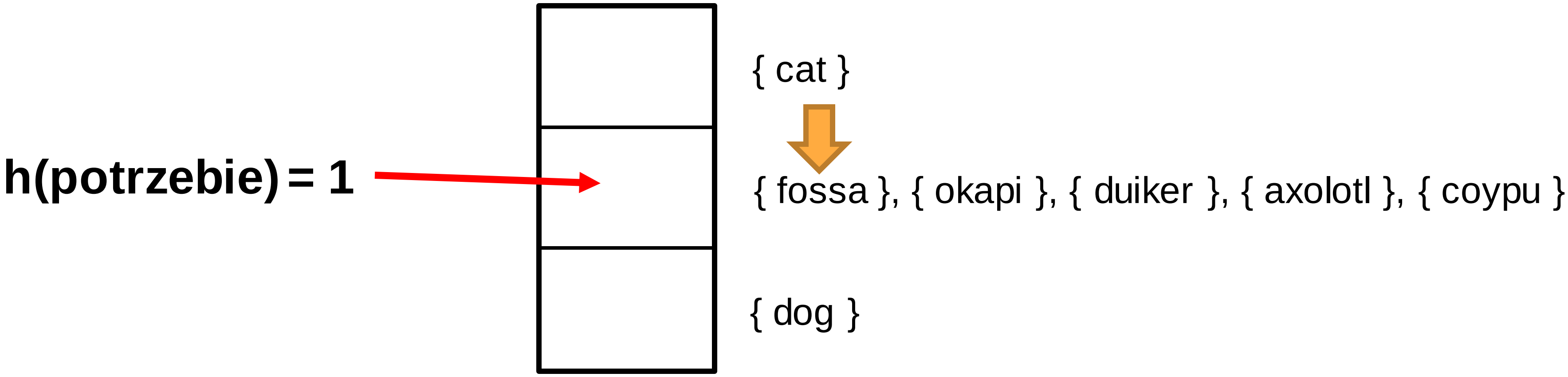
find(potrzebie)



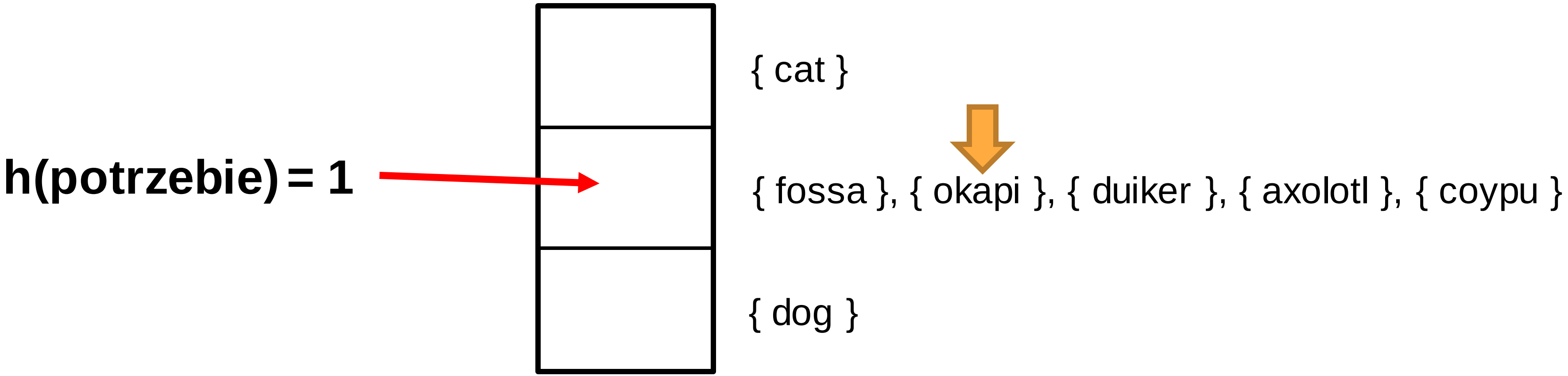
Hash Table with Chaining



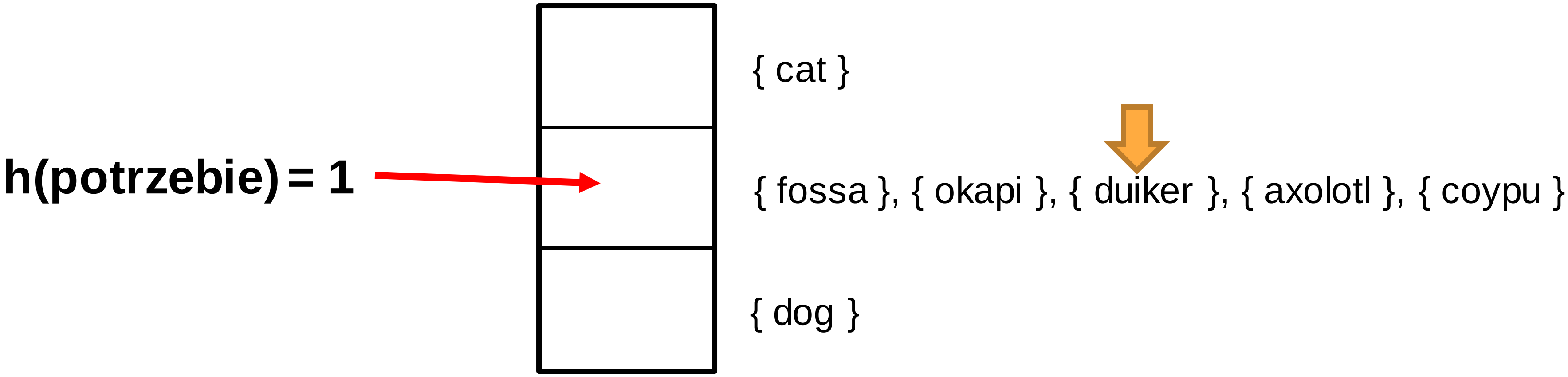
Hash Table with Chaining



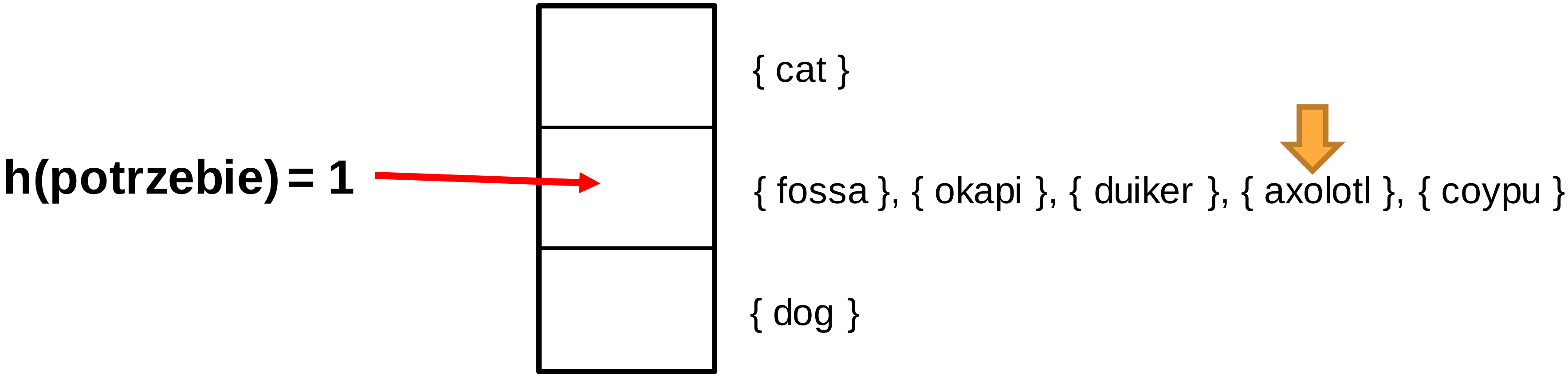
Hash Table with Chaining



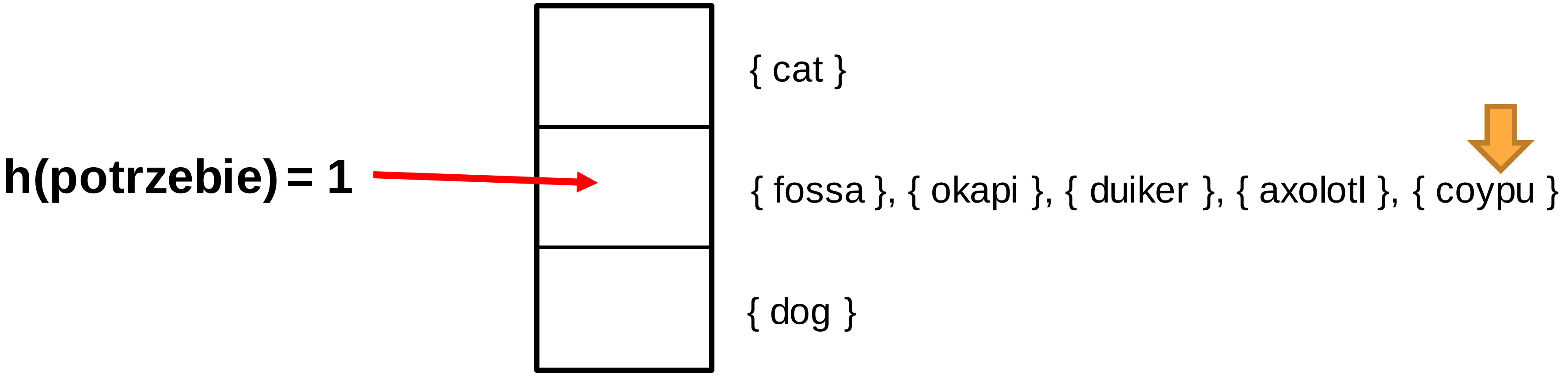
Hash Table with Chaining



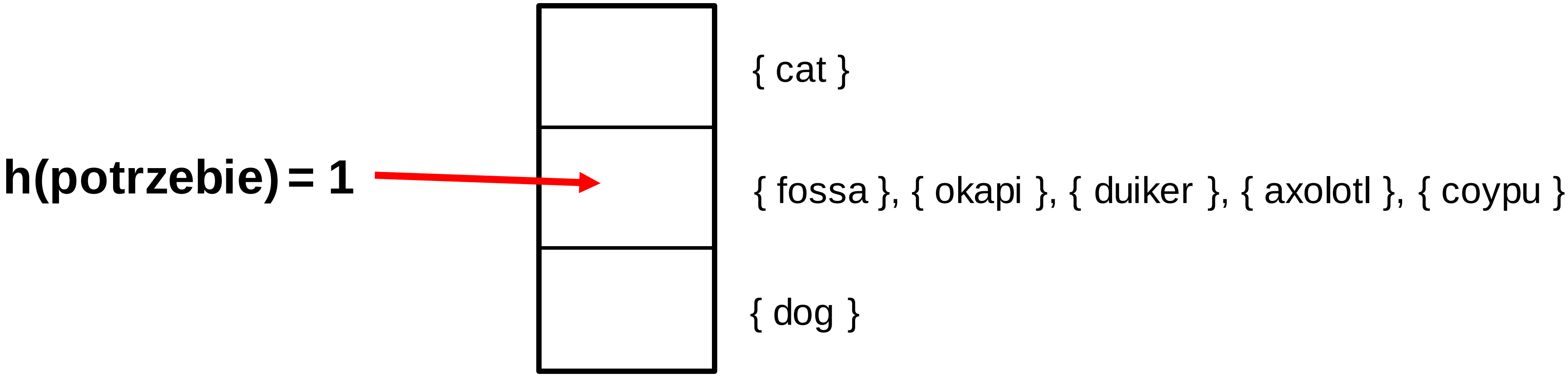
Hash Table with Chaining



Hash Table with Chaining



Hash Table with Chaining



NOT FOUND

What is Performance of a Hash Table?

- “Performance” = “cost to do a find()”

- (remove, and *maybe* insert, similarly traverse list for some bucket)

What is Performance of a Hash Table?

- “Performance” = “cost to do a find()”

- (remove, and *maybe* insert, similarly traverse list for some bucket)
    - Insert traverses list if we must check for duplicates

What is Performance of a Hash Table?

- “Performance” = “cost to do a find()”
- (remove, and *maybe* insert, similarly traverse list for some bucket)
- Suppose table holds n records
- *In worst case*, all n records hash to one bucket
- Searching this bucket takes time **???**

What is Performance of a Hash Table?

- “Performance” = “cost to do a find()”
- (remove, and *maybe* insert, similarly traverse list for some bucket)
- Suppose table holds n records
- *In worst case*, all n records hash to one bucket
- Searching this bucket takes time **$\Theta(n)$**

Cost of Hash Table (Worst-Case)

- Time complexities for dictionary operations

Structure	insert	delete	find	space
unsorted list	Θ(1)	Θ(n)	Θ(n)	Θ(n)
sorted list	Θ(n)	Θ(n)	Θ(n)	Θ(n)
sorted array	Θ(n)	Θ(n)	Θ(log n)	Θ(n)
hash table	**Θ(n)**	**Θ(n)**	**Θ(n)**	**Θ(m+n)**

I thought the point was to get sublinear-time ops!

A Weaker Performance Estimate

- **Assume** that, given a key k in U, hash function h is equally likely to map k to each value in [0, m), independent of all other keys.
- This assumption is called **Simple Uniform Hashing**.
- Now suppose we hash n keys $k_1 \ldots k_n$ from U into the table, then call find(k*) for some key k*.
- What is the **average [over choice of keys] cost** to search the table for k*?

Average Cost of Search

- Total of n elements distributed over m slots

- Average size of bucket is therefore…

Average Cost of Search

- Total of n elements distributed over m slots

- Average size of bucket is therefore… **n/m**

Average Cost of Search

- Total of n elements distributed over m slots
- Average size of bucket is therefore… **n/m**
- Suppose k* is *not* in the table.
- Cost of find(k*) is Θ(1) to compute h(k*), plus Θ(bucket size) to search
- *h(k*) equally likely to be any bucket*, so average cost of unsuccessful find is **Θ(1 + n/m)**.

Average Cost of Search

- Total of n elements distributed over m slots
- Average size of bucke
- Suppose k* is *not* in t
- Cost of find(k*) is Θ(1 … search
- *h(k*) equally likely to be any bucket*, so average cost of unsuccessful find is **Θ(1 + n/m)**.

Follows from Simple Uniform Hashing

Average Cost of Search

- Average cost of unsuccessful find is **$\Theta(1 + n/m)$**.
- Similar arguments from SUH show that average cost of *successful* find is also **$\Theta(1 + n/m)$**.
- **Defn**: $\alpha = n/m$ is called the **load factor** of the hash table.

Cost of Hash Table (Average Under SUH)

- Time complexities for dictionary operations

Structure	insert	delete	find	space
unsorted list	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
sorted list	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
sorted array	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
hash table	**$\Theta(1 + \alpha)$**	**$\Theta(1 + \alpha)$**	**$\Theta(1 + \alpha)$**	**$\Theta(m+n)$**

Load factor determines performance of hash table

Controlling the Load Factor

- If we know that the table will hold at most n records…
- We can make # of buckets m proportional to n, say m=cn. (e.g. c=0.75)
- This choice makes our load factor n/m a constant (called α).
- **Ex**: if we set m = n/4, load factor α is 4.
- But then expected search cost is $\Theta(1 + \alpha) = \mathbf{\Theta(1)}$.

Cost of Hash Table (Average Under SUH, m = cn)

- Time complexities for dictionary operations

Structure	`insert`	`delete`	`find`	`space`
unsorted list	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
sorted list	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
sorted array	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
hash table	**$\Theta(1)$**	**$\Theta(1)$**	**$\Theta(1)$**	**$\Theta(n)$**

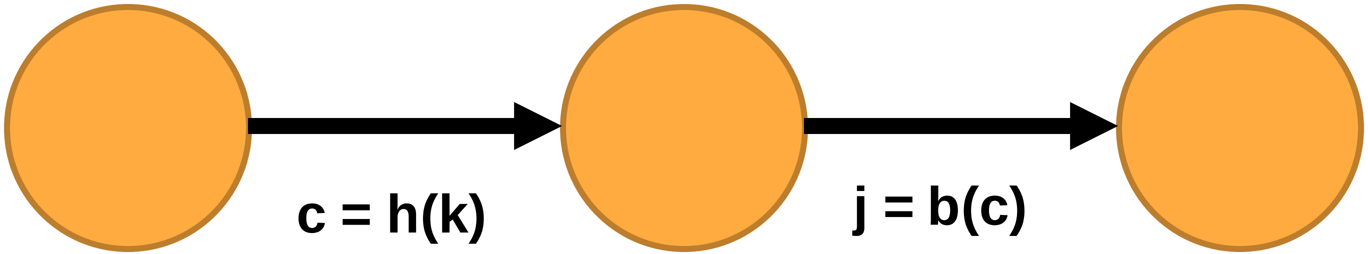
Hashing gives expected constant-time dictionary ops in linear space!

How Do We Approach Ideal Performance?

- Hash function h(k) must approximate SUH assumptions
- Must distribute keys equally, independently across range [0, m)
- [We need to talk about how to **design** a good hash function h(k)!]
- Moreover, input keys we see must have “average” behavior
- *(Alternative: attacker with knowledge of h(k) chooses keys so as to elicit worst-case behavior from your table!)*

And Now, Some Hash Function Design

Hash Function Pipeline – Two Steps

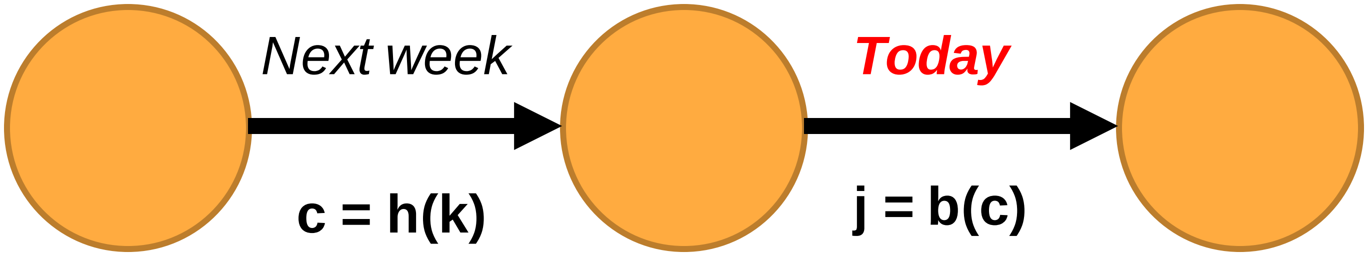


Objects
(keys k)

Integers
(hashcodes c)

Buckets
(indices j)

Hash Function Pipeline – Two Steps



Objects
(keys k)

Integers
(hashcodes c)

Buckets
(indices j)

Assumptions

- Objects to be hashed have been converted to integer hashcodes
- Hashcodes are in range [0, N)
- Need to convert hashcodes to indices in [0, m) ← *m = table size*

Assumptions

- Objects to be hashed have bee
- Hashcodes are in range [0, N)
- Need to convert hashcodes to i

NB: Java hashcodes can be positive *or* negative. May need to take absolute value or otherwise make ≥ 0!

Goals for Mapping to Indices (from SUH)

- Each hashcode should be about equally likely to map to any value in [0, m).

- Mappings for different hashcodes should be independent, hence uncorrelated – knowing the mapping for one should give little or no information about the mapping for another.

Two Main Approaches to Index Mapping

- Division hashing
- Multiplicative hashing
- (Other strategies exist; beyond scope of 247)

Division Hashing

- b(c) = c mod m
- *“bucket index = hashcode modulo table-size”*
- Very easy to implement (mod in Java is %)
- Result is surely in range [0, m) (***if*** *c is non-negative*!)

The Perils of Division Hashing

- Does every choice of m yield SUH-like behavior?
- **Ex**: Suppose that m is divisible by a small integer d.
- **Claim**: if j = c mod m, then j mod d = c mod d
- *So what?*

The Perils of Division Hashing

- **Ex**: Suppose that m is divisible by a small integer d.
- **Claim**: if j = c mod m, then j mod d = c mod d
- *E.g., if d = 2, then even hashcodes map to even indices.*
- *"Natural" subsets of all hashcodes do not map uniformly across the entire table → not SUH behavior!*

The Perils of Division Hashing (Proof)

- **Claim**: if $j = c \bmod m$, then $j \bmod d = c \bmod d$
- *Pf:* Suppose $c = x + ym$.
- Since $d \mid m$, $c = x + zd$ for some z.
- Hence $c \bmod d = x = (c \bmod m) \bmod d = j \bmod d$. QED

A Particularly Bad Case

- **Ex**: Suppose that $m = 2^v$
- Hashcodes with same v low-order bits map to same index

10011010111101100101111000010101

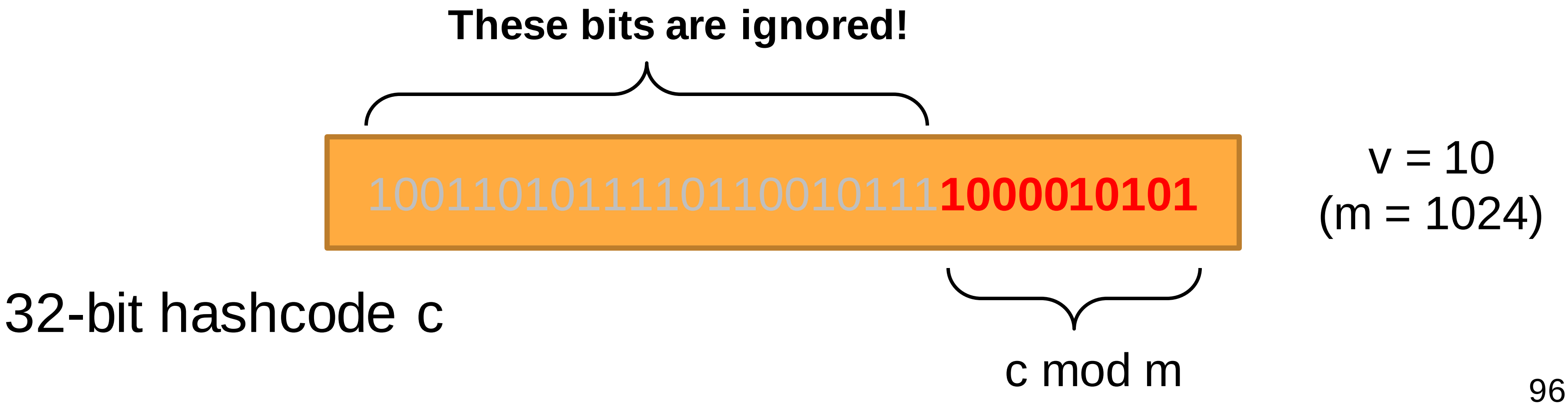
32-bit hashcode c

A Particularly Bad Case

- **Ex**: Suppose that $m = 2^v$
- Hashcodes with same v low-order bits map to same index

A Particularly Bad Case

- **Ex**: Suppose that $m = 2^v$
- Hashcodes with same v low-order bits map to same index



Advice on Division Hashing

- Table size m should be chosen so that
  - No (*obvious*) correlations between hashcode bit pattern and index
  - Index depends on **all** bits of hashcode, not just some

- *Idea*: make m a prime number (no small factors)

- *Avoid choices of m close to powers of 2 or 10*

What's Wrong with m Near Power of 2 or 10?

- **Ex**: Suppose $m = 2^v - 1$
- If $c = c_0 + 2^v c_1 + 2^{2v} c_2 + 2^{3v} c_3 + \ldots.$
- $c \bmod m = c_0 + c_1 + c_2 + c_3 + \ldots. \bmod m$
- Could permute chunks of v bits in c and get same index!
- *(Think about strings encoded using v bits per character)*

Other Thoughts on Division Hashing

- The operation "c mod m" is expensive on most computers
- (unless m is a *constant* known at compile time)
- Modulo op is most efficient when m is a power of 2… but this is a poor choice for division hashing!

Two Main Approaches to Index Mapping

- Division hashing
- Multiplicative hashing
- (Other strategies exist; beyond scope of 247)

Multiplicative Hashing

- Let A be a *real number* in [0, 1).
- $\boldsymbol{b(c) = \left\lfloor \left((c \cdot A) \; mod \; 1.0 \right) \cdot m \right\rfloor}$
- "x mod 1.0" means "fractional part of x."
- E.g. 47.2465 mod 1.0 = 0.2465
- cA mod 1.0 is in [0, 1), so b(c) is an integer in [0, m) – an index!

Initial Observations

- A should not be *too* small – would map many hashcodes to 0.
- → Suggest picking A from [0.5, 1)
- If q = cA mod 1.0 is distributed uniformly in [0, 1), then we can use *any* value for m and still get uniform indices.
- In particular, we can use $m = 2^v$ if we want.

Why Is Multiplication a Good Hashing Strategy?

- Mapping c → q = cA mod 1.0 is a *diffusing operation*

- I.e., most significant digits of q depend (in a complex way) on many digits of c. (Makes q looks uniform, obscures correlations among c's.)

- Hence, bin number $\lfloor q \cdot m \rfloor$ looks uniform, uncorrelated with c.

- (Same is true if we replace "digits" by "bits" and work in binary)

Example of Diffusion

```
  1234
x0.6734
-------
```

Assumed:

- Integer c has fixed some # of digits
- We use same # of digits of A after decimal

Example of Diffusion

```
  1234
x0.6734
-------
 .4936
```

Example of Diffusion

```
   1234
x0.6734
-------
  .4936
 3.7020
```

Example of Diffusion

```
    1234
 x0.6734
--------
   .4936
  3.7020
 86.3800
```

Example of Diffusion

```
     1234
  x0.6734
---------
    .4936
   3.7020
  86.3800
 740.4000
```

Example of Diffusion

```
      1234
   x0.6734
----------
     .4936
    3.7020
   86.3800
 +740.4000
```

- First digit after decimal is middle digit of product
- Middle digits depend on all (or most) digits of c and all or most digits of A
- These digits determine bin number

Is Every Choice of A Equally Good?

- Not all A's have equally good diffusion/complexity properties.
- Fractions with few nonzero digits (e.g. 0.75) or repeating decimals (e.g. 7/9 = 0.7777777…..) have poor diffusion and/or low complexity.
- **Advice: pick an irrational number between 0.5 and 1.**
- **Ex:** $\boldsymbol{A = \frac{\sqrt{5}-1}{2}}$ $\approx$ 0.6180339887498948482045868343656 [Knuth]

Multiplication Hashing Without Floating-Point Math

- What if you can't / don't want to use floating-point math?
- (May be more expensive than integer math)
- If we know our hashcodes c have at most **d** digits, we can multiply A by 10^d initially and do everything we need using only integer arithmetic.
- Similarly, if hashcodes have at most **w** bits, we can multiply A by 2^w initially.
- This trick is called "**fixed-point arithmetic**".

Previous Example, in Fixed-Point Decimal

```
   1234
x 0.6734
--------
```

Assumed:

- Integer c has at most 4 digits
- We use same # of digits of A after decimal

Previous Example, in Fixed-Point Decimal

```
      1234
x     6734        ÷ 10^4
```

$$\begin{array}{r} 1234 \\ \times\ 6734 \\ \hline \end{array} \quad \div\ 10^4$$

(multiply, but remember how to undo)

Previous Example, in Fixed-Point Decimal

```
      1234
x     6734        ÷ 10^4
----------
      4936
```

Previous Example, in Fixed-Point Decimal

```
      1234
x     6734        ÷ 10^4
----------
      4936
     37020
```

Previous Example, in Fixed-Point Decimal

```
      1234
x     6734        ÷ 10^4
----------
      4936
     37020
    863800
```

Previous Example, in Fixed-Point Decimal

```
       1234
x      6734        ÷ 10^4
-----------
       4936
      37020
     863800
    7404000
```

Previous Example, in Fixed-Point Decimal

```
     1234
x    6734
---------
     4936
    37020
   863800
+ 7404000
---------
  8309756
```

$\div\ 10^4$

We know decimal point goes here

cA mod 1 = **9756** $\div$ **10^4**

Index Computation in Fixed-Point Decimal

- Suppose m = 100 = 10^2.

- (cA mod 1) m = **9756 ÷ 10^4 x 10^2**
- = 9756 ÷ **10^{4-2}**
- = 9756 ÷ **10^2**

Index Computation in Fixed-Point Decimal

- Suppose m = 100 = 10^2.

- (cA mod 1) m = **9756** $\div$ $\mathbf{10^4 \times 10^2}$
- = 9756 $\div$ $\mathbf{10^{4-2}}$
- = 9756 $\div$ $\mathbf{10^2}$

Again, we know decimal point goes here

Index Computation in Fixed-Point Decimal

- Suppose m = 100 = 10^2.

- (cA mod 1) m = **9756 ÷ 10^4 x 10^2**
- = 9756 ÷ **10^{4-2}**
- = 9756 **÷ 10^2**

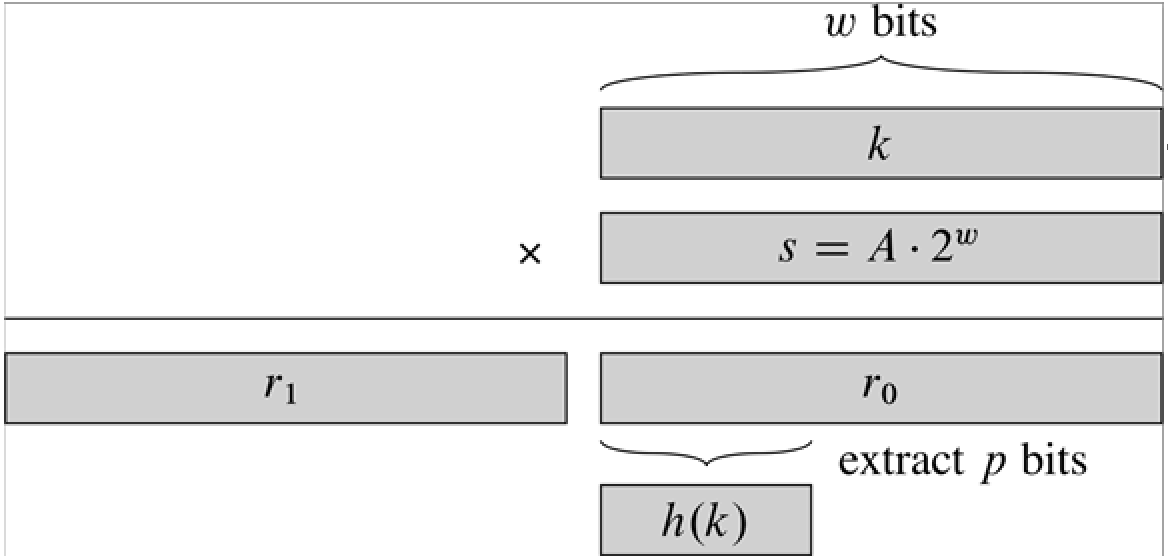
Again, we know decimal point goes here

- **Hence,** $\lfloor ((c \cdot A) \bmod 1.0) \cdot m \rfloor = 97$

What About Fixed-Point Binary?

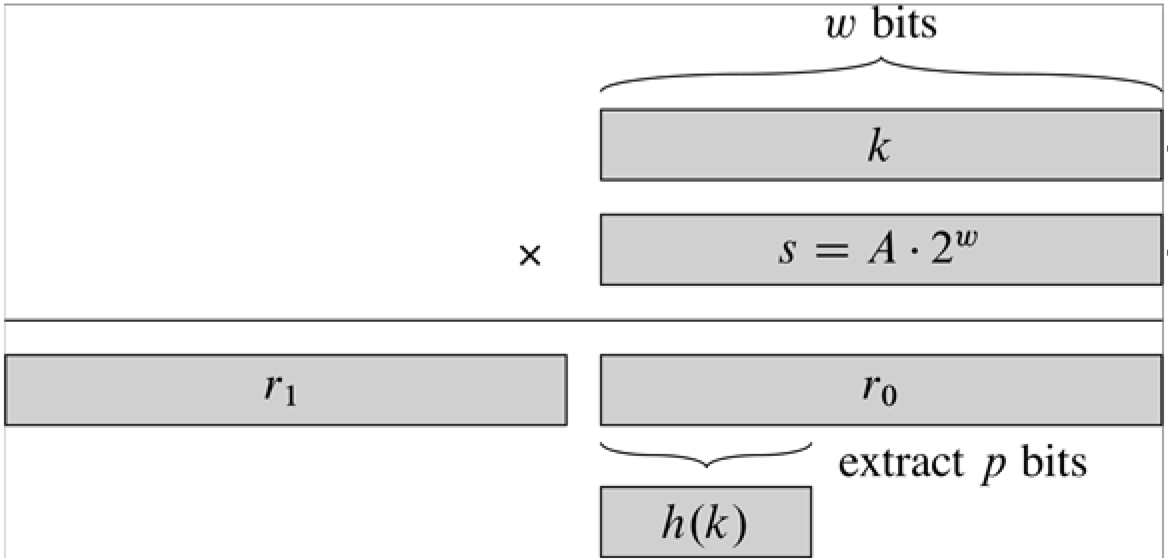
- Book presents the binary version.
- It’s also how you would typically implement it on a computer!
- If you have had 132, then the following slides will make more sense
  - If not, follow along as best you can, and look at this again after you’ve had 132

For base 2 (let’s assume w = 32)



result of `.hashCode()`

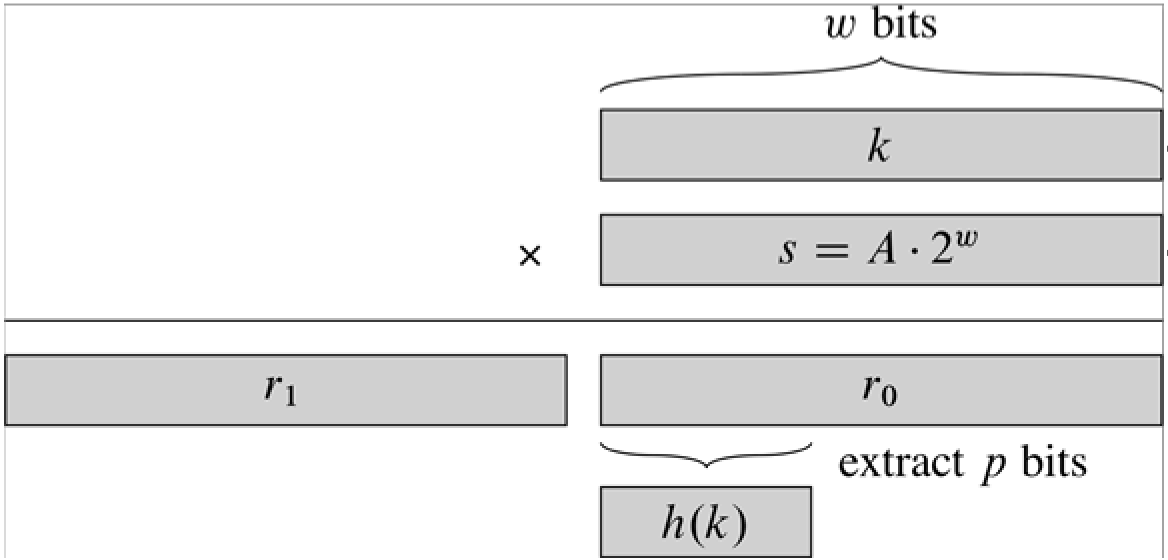
For base 2 (let’s assume w = 32)



result of `.hashCode()`

A shifted left by 32 bits

For base 2 (let's assume w = 32)

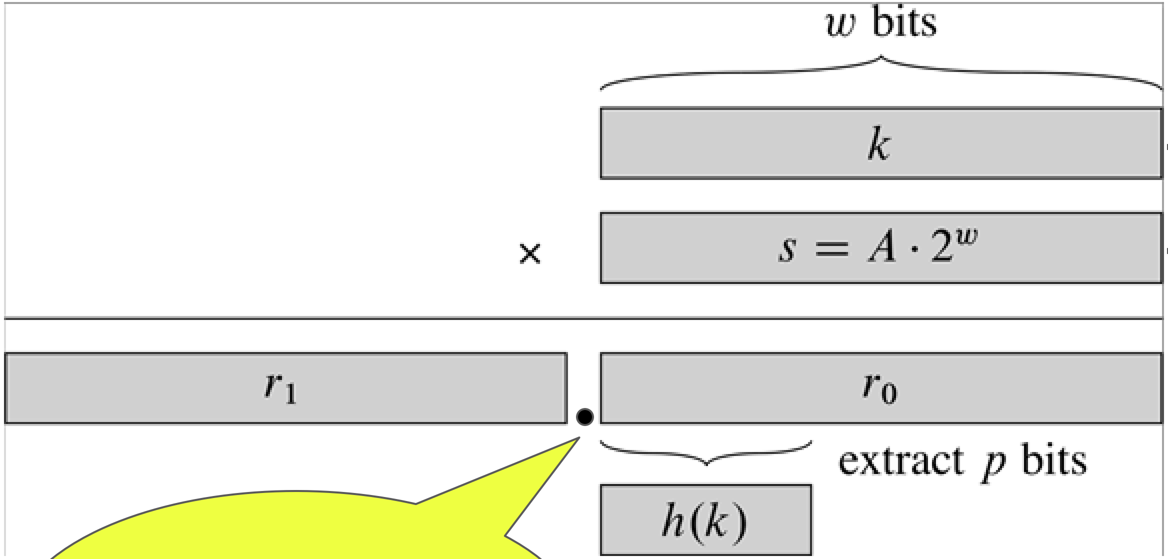


result of `.hashCode()`

A shifted left by 32 bits

The product of two w-bit numbers yields a 2w-bit result

For base 2 (let's assume w = 32)

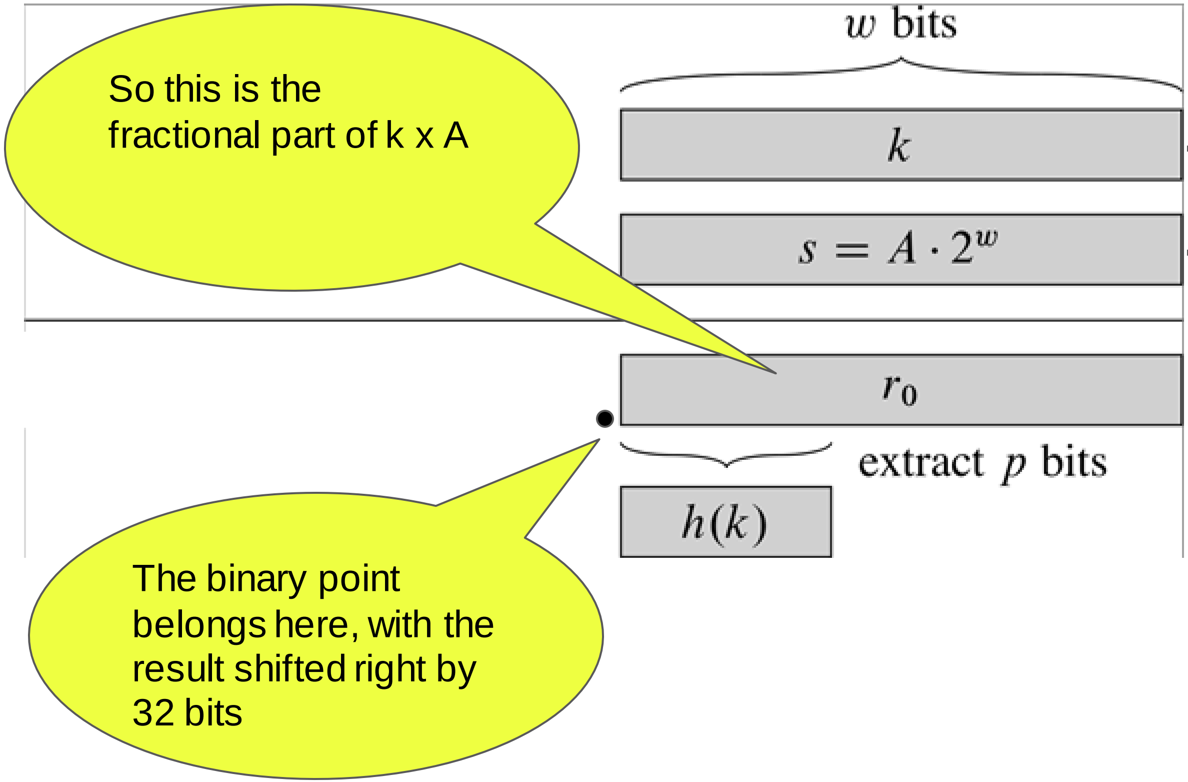


result of `.hashCode()`

A shifted left by 32 bits

The binary point belongs here, with the result shifted right by 32 bits

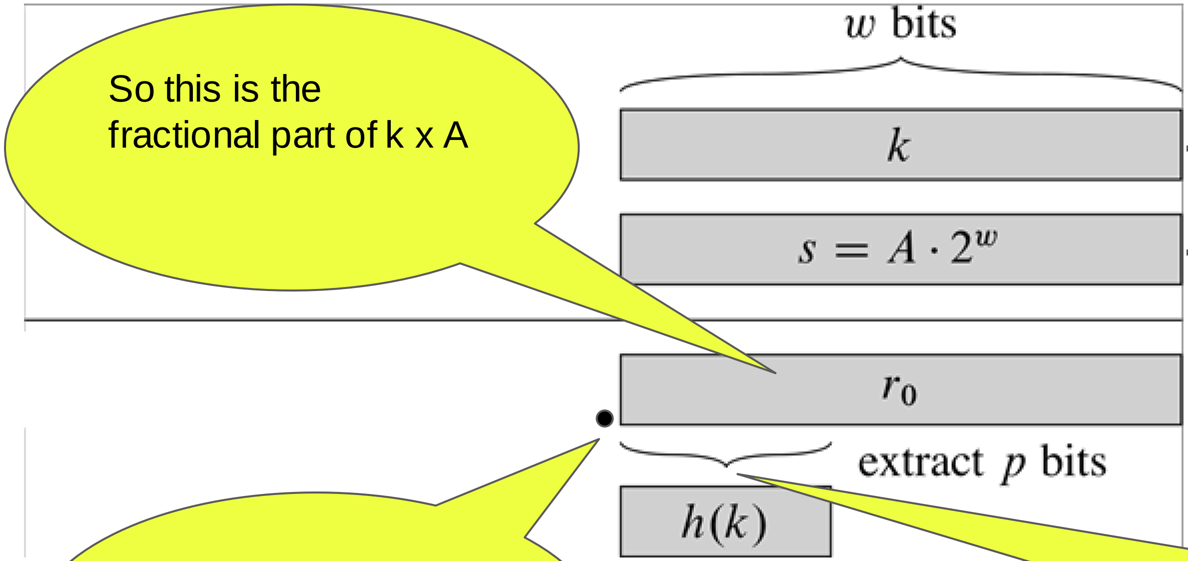
For base 2 (let’s assume w = 32)



result of `.hashCode()`

A shifted left by 32 bits

For base 2 (let’s assume w = 32)



So this is the fractional part of k x A

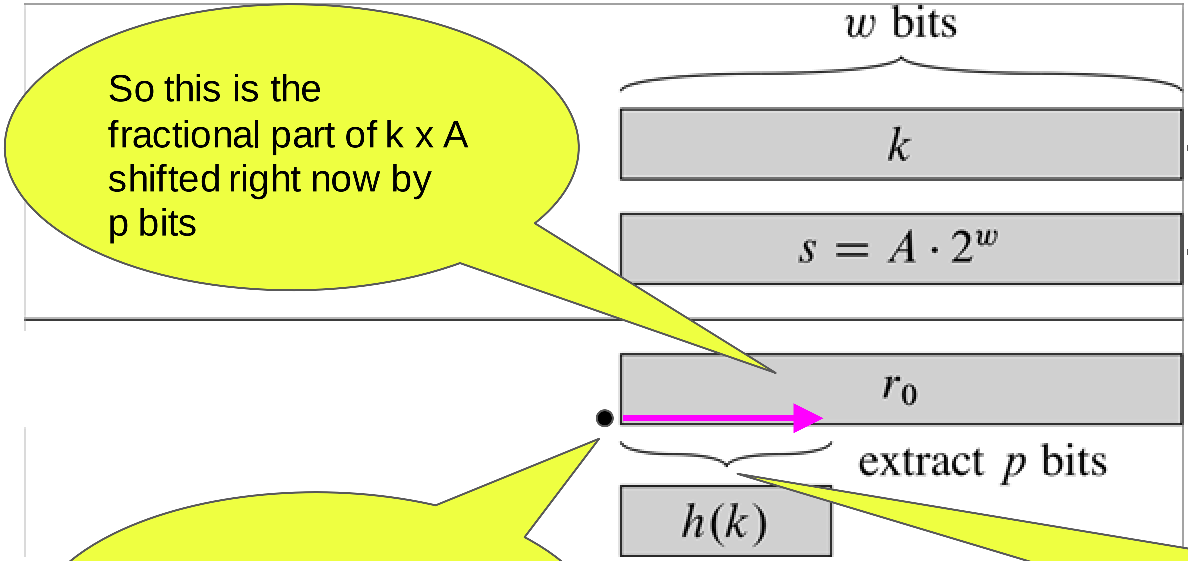
result of `.hashCode()`

A shifted left by 32 bits

The binary point belongs here, with the result shifted right by 32 bits

If $m = 2^p$ then multiplying the fractional part by m yields these p bits

For base 2 (let’s assume w = 32)



So this is the fractional part of k x A shifted right now by p bits

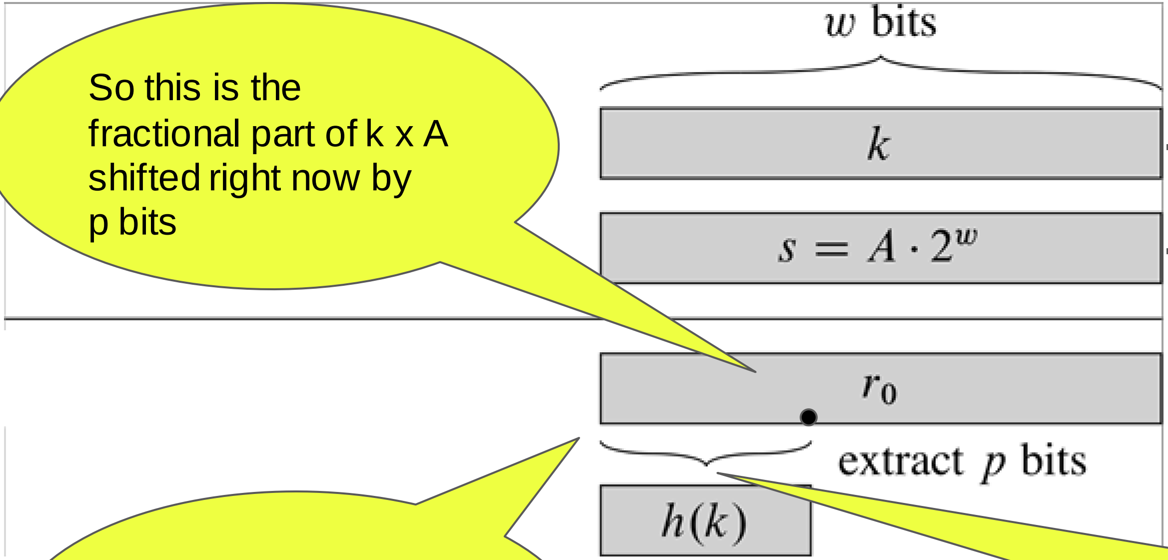
result of `.hashCode()`

A shifted left by 32 bits

The binary point belongs here, with the result shifted right by 32 bits

If m = 2^p then multiplying the fractional part by m yields these p bits

For base 2 (let’s assume w = 32)



result of `.hashCode()`

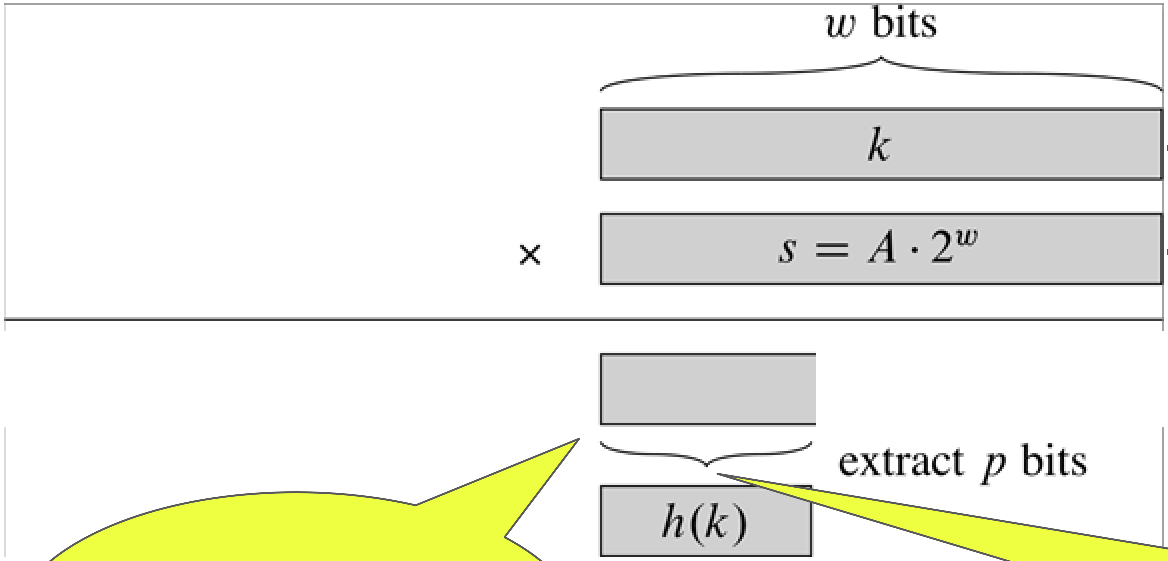
A shifted left by 32 bits

So this is the fractional part of k x A shifted right now by p bits

The binary point belongs here, with the result shifted right by 32 bits

If m = 2^p then multiplying the fractional part by m yields these p bits

For base 2 (let’s assume w = 32)



result of `.hashCode()`

A shifted left by 32 bits

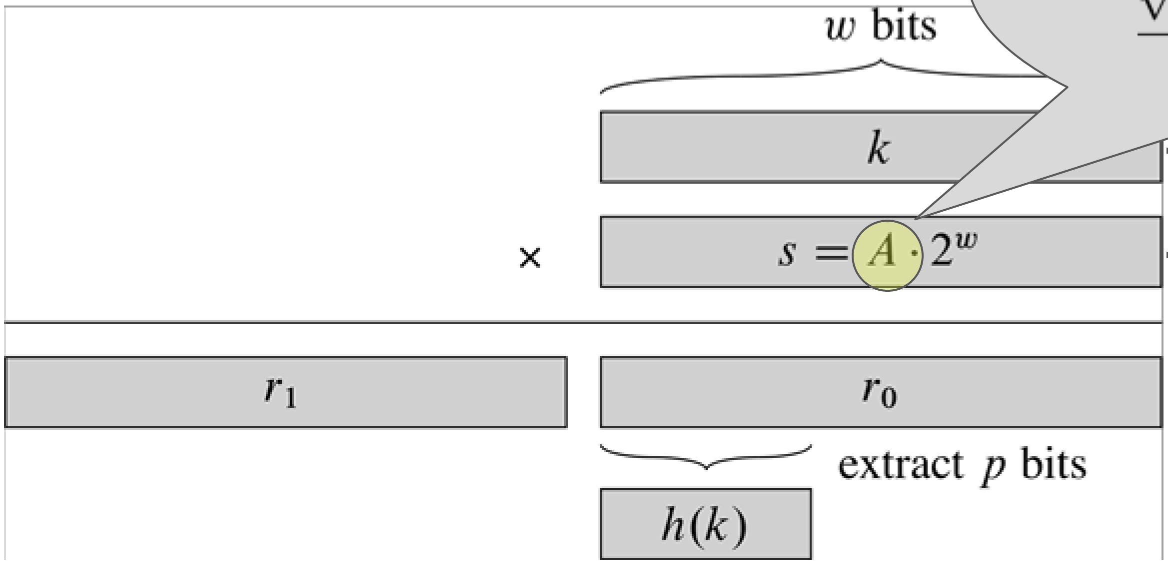
The binary point belongs here, with the result shifted right by 32 bits

If m = 2^p then multiplying the fractional part by m yields these p bits

For base 2 (let's assume w =

Assume we use Knuth's A:

$$\frac{\sqrt{5}-1}{2} \approx 0.6180339887\ldots$$

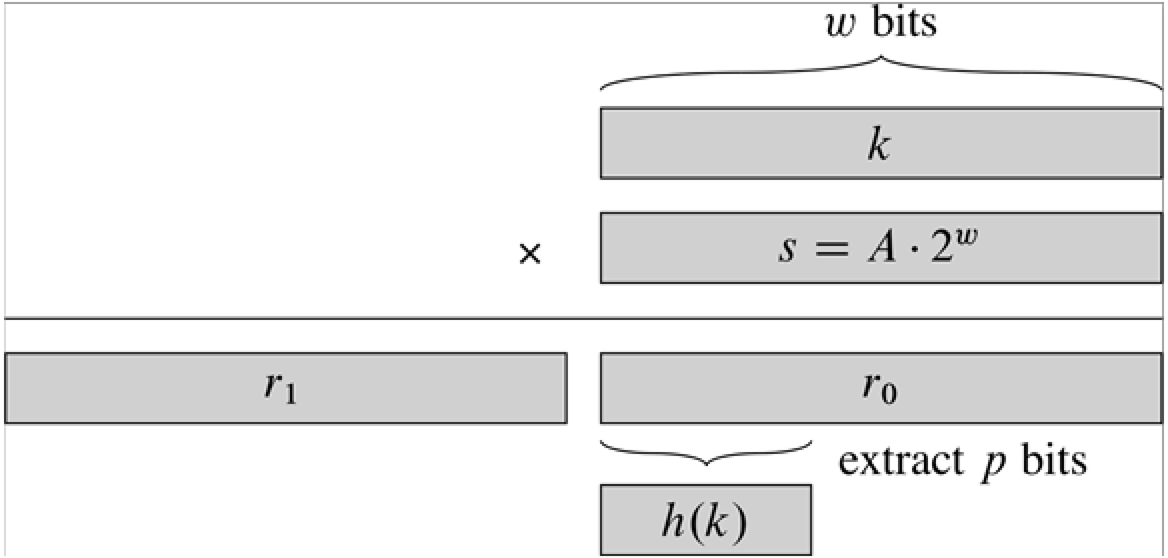


result of `.hashCode()`

A shifted left by 32 bits

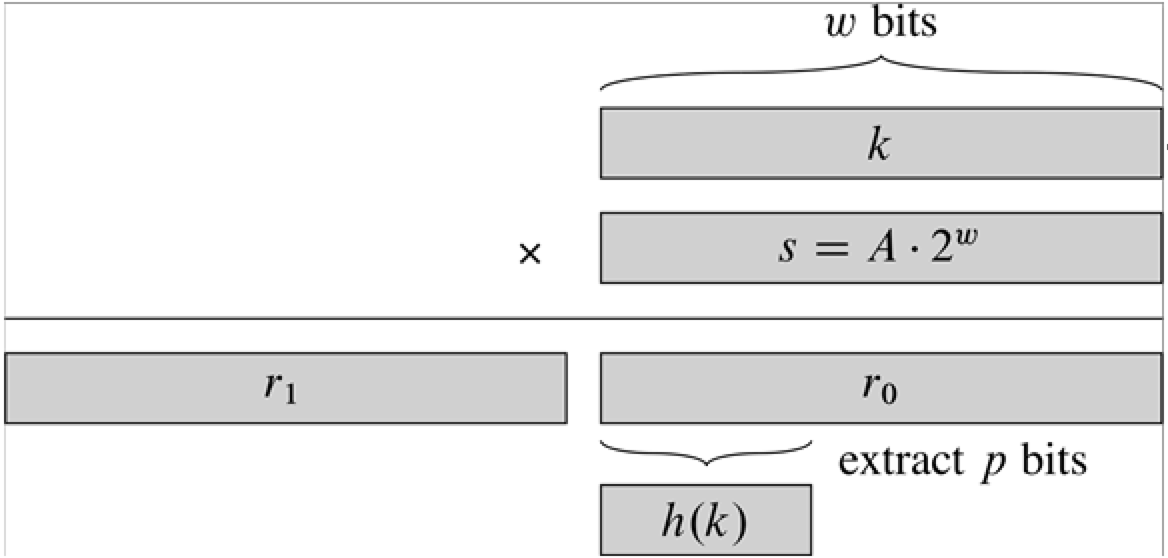
Example (page 264 in text)

w = 32
p = 14 → m = 16384



Example (page 264 in text)

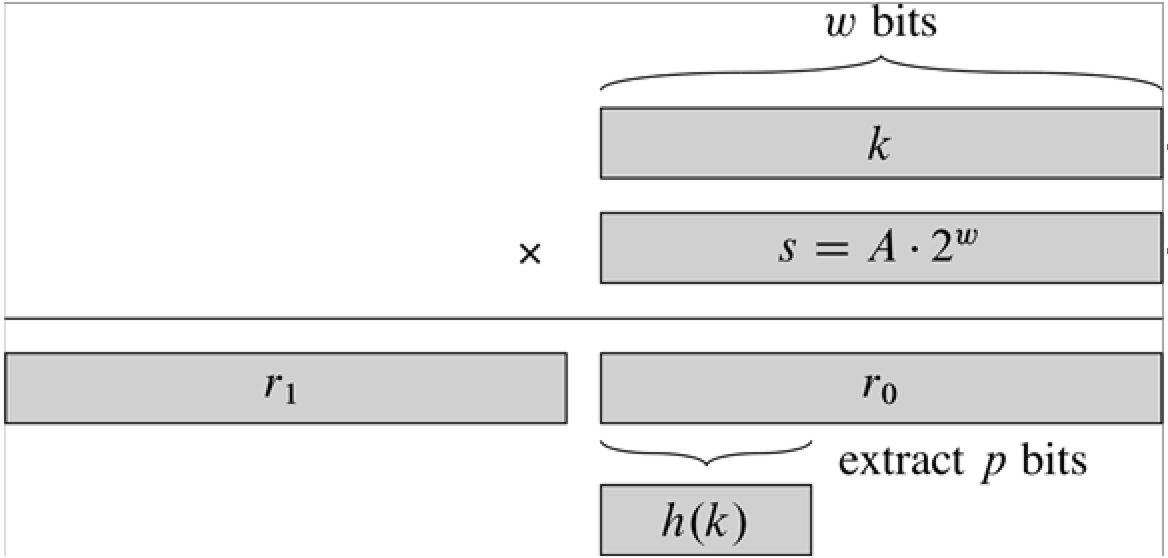
w = 32
p = 14 → m = 16384



k = 123456

Example (page 264 in text)

w = 32
p = 14 → m = 16384

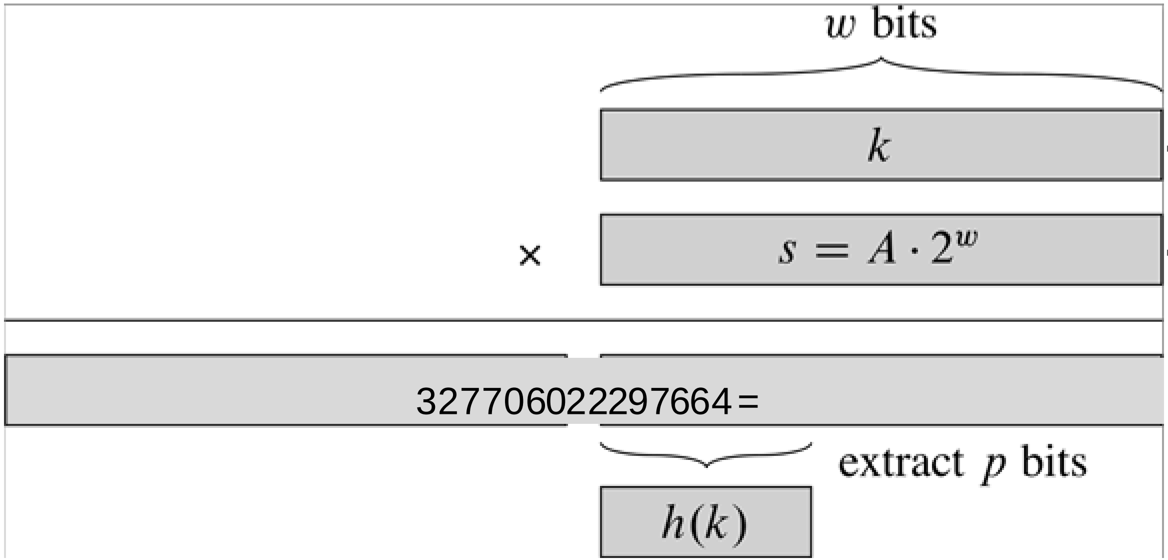


k = 123456

A x 2^{32} = 2654435769

Example (page 264 in text)

w = 32
p = 14 → m = 16384

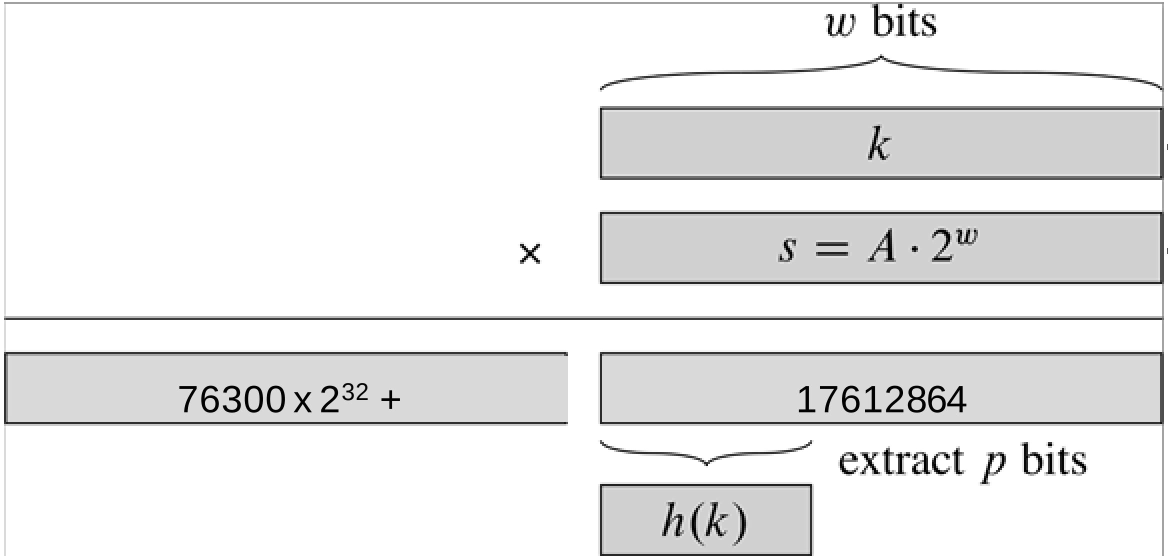


k = 123456

A x 2^{32} = 2654435769

Example (page 264 in text)

w = 32
p = 14 → m = 16384

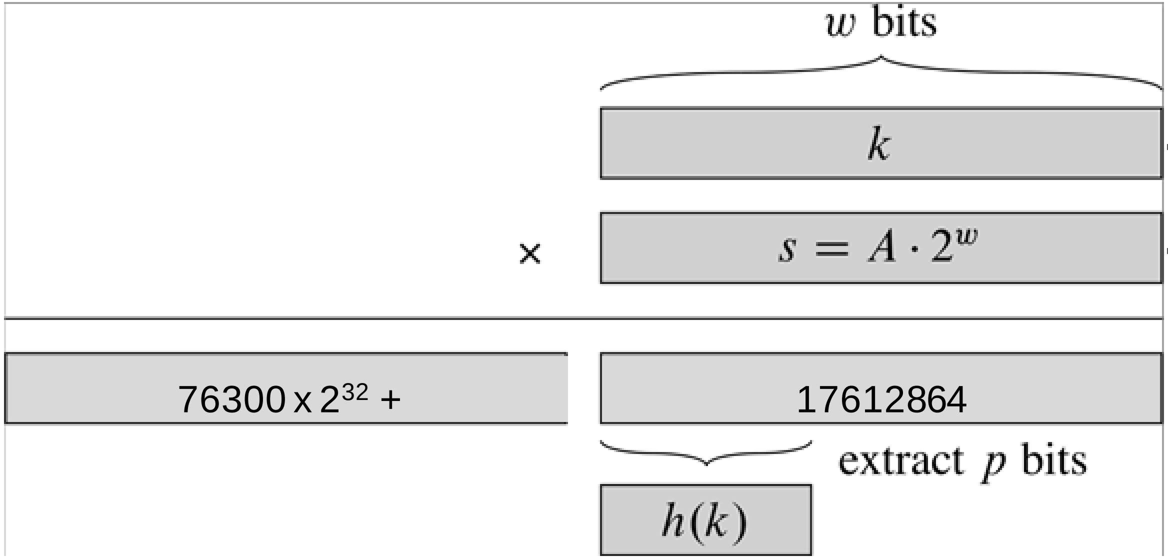


k = 123456

A x 2^{32} = 2654435769

Example (page 264 in text)

w = 32
p = 14 → m = 16384



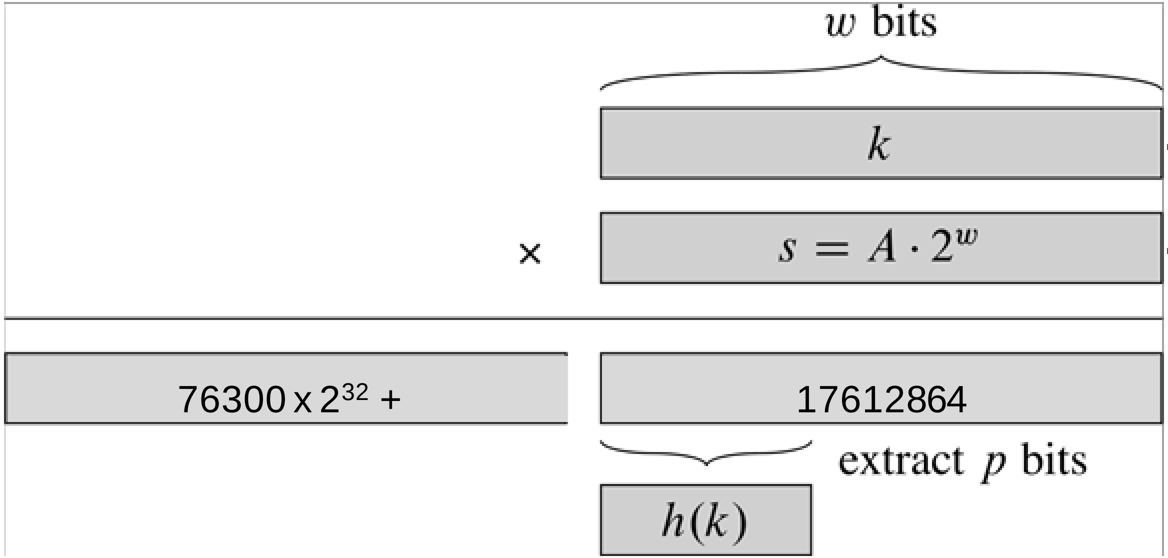
k = 123456

A x 2^{32} = 2654435769

32 bit representation for 17612864 is
01 0C C0 40

Example (page 264 in text)

w = 32
p = 14 → m = 16384



k = 123456

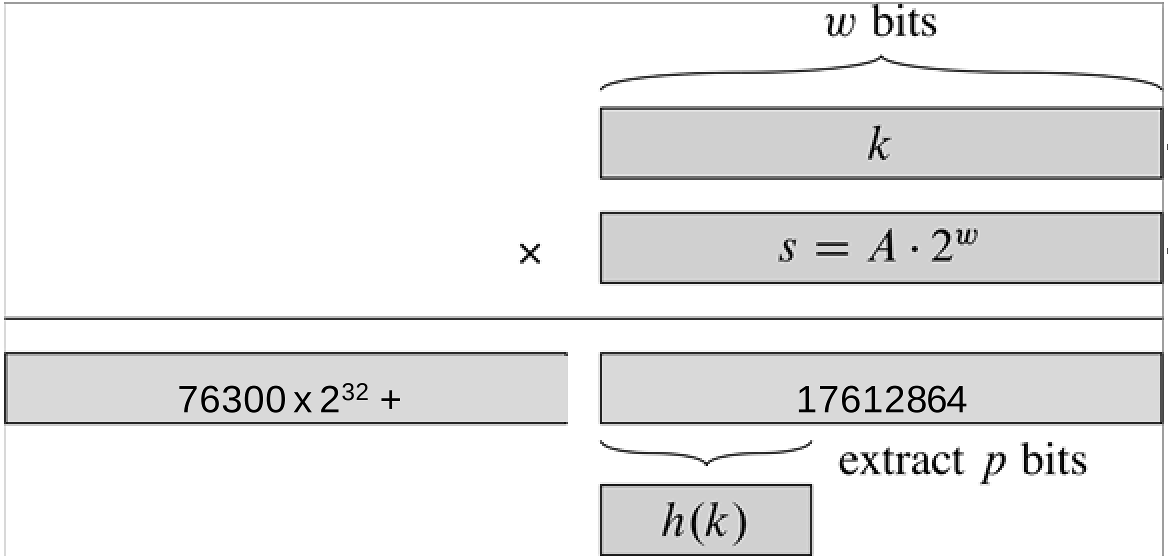
A x 2^{32} = 2654435769

Top 14 bits
0
0000

32 bit representation for 17612864 is
01 0C C0 40

Example (page 264 in text)

w = 32
p = 14 → m = 16384



k = 123456

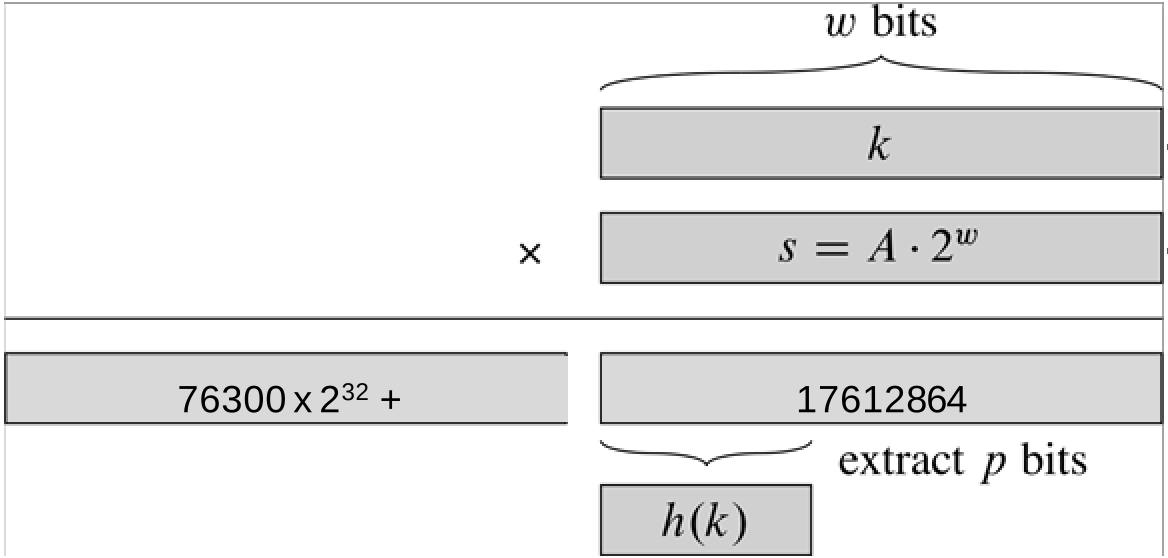
A x 2^{32} = 2654435769

Top 14 bits
01
0000 0001

32 bit representation for 17612864 is
01 0C C0 40

Example (page 264 in text)

w = 32
p = 14 → m = 16384



k = 123456

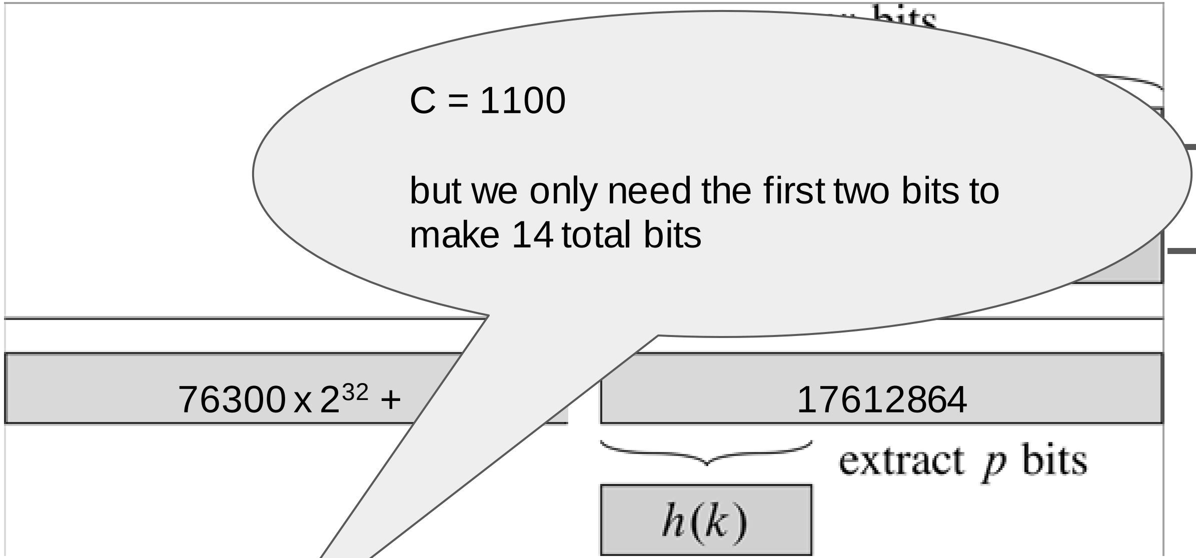
A x 2^{32} = 2654435769

Top 14 bits
01 0
0000 0001 0000

32 bit representation for 17612864 is
01 0C C0 40

Example (page 264 in text)

w = 32
p = 14 → m = 16384



k = 123456

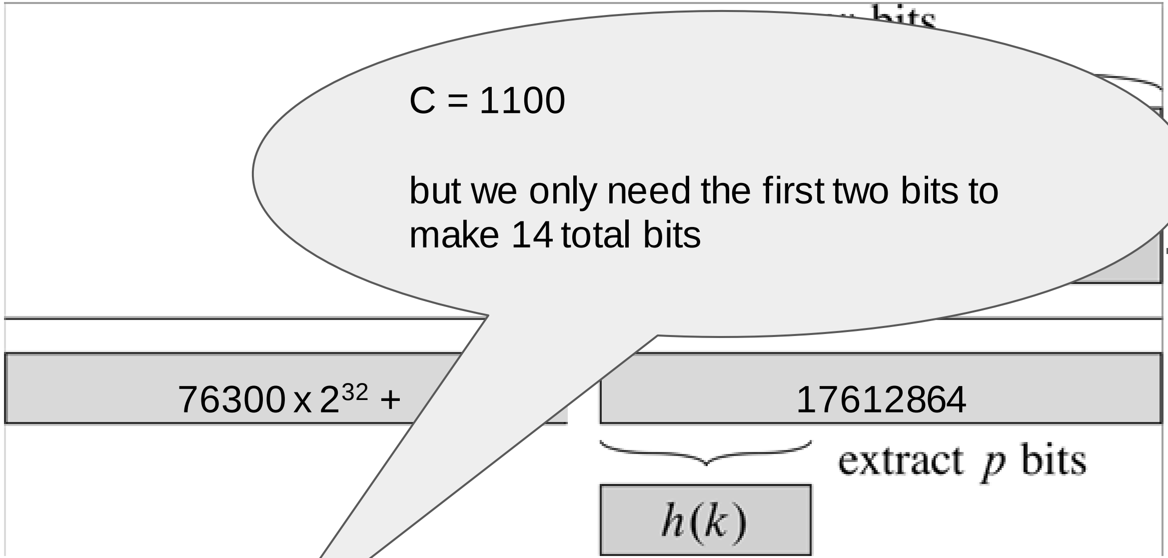
A x 2^32 = 2654435769

Top 14 bits
01 0C
0000 0001 0000

32 bit representation for 17612864 is
01 0C C0 40

Example (page 264 in text)

w = 32
p = 14 → m = 16384



C = 1100

but we only need the first two bits to make 14 total bits

k = 123456

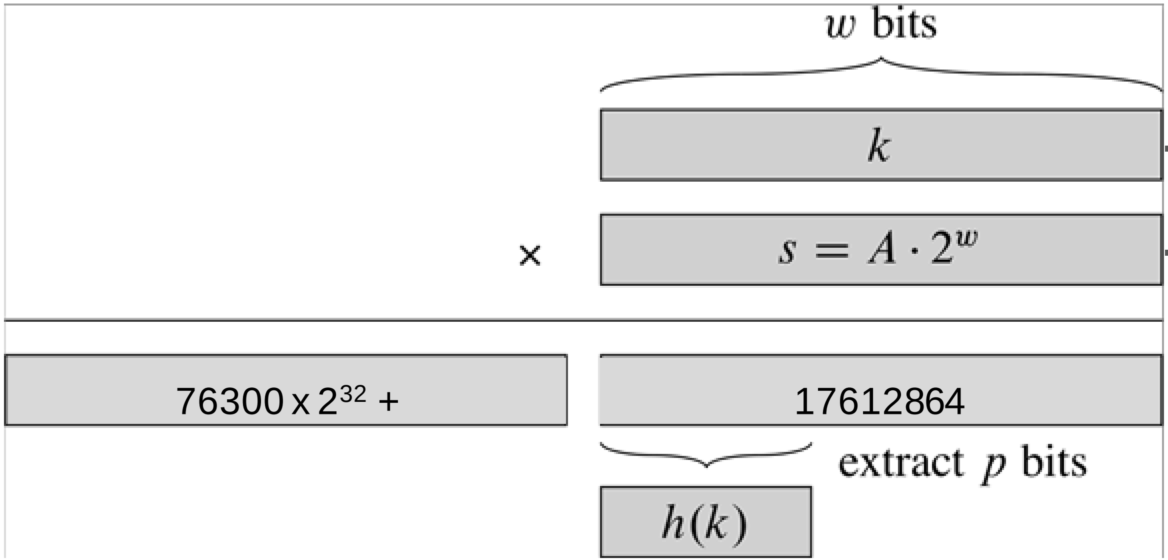
A x 2^{32} = 2654435769

Top 14 bits
01 0C
0000 0001 0000 11

32 bit representation for 17612864 is
01 0C C0 40

Example (page 264 in text)

w = 32
p = 14 → m = 16384



k = 123456

A x 2^{32} = 2654435769

hashes to bucket 67

Top 14 bits
01
0000 0001 0000 11 = 64 + 3 = 67

32 bit representation for 17612864 is
01 0C C0 40

A Good Implementation

- Choose $m = 2^p$ buckets
- Assume .hashCode() yields 32-bit *unsigned* integer k [*does not exist in Java*]
- *Pre-compute the constant* $s = 2^{32} \times A$
- Assume that if sk overflows 32 bits, we get only lower 32 bits of result
- Index computation on input k is then $sk \div 2^{32-p}$ = sk >> (32 – p)
- *This is a close relative of the function you'll play with in Studio 7.*

End of Lecture 7