Lecture 9: More About Hashing

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

Announcements

- **Lab 7** – pre-lab due tonight, code/post-lab due Friday
  - Please remember to commit AND push AND check bitbucket.org!
  - Please remove debugging code!

- Exam reschedule requests for Exam 2 and Exam 3
  - Due next Tuesday 11:59 pm
  - Form here on website (will be removed after next Tuesday)

Agenda for today

- *Leftover hash*… finish up multiplication hashing
- A second strategy for hash table design – open addressing
- How to map objects to hashcodes

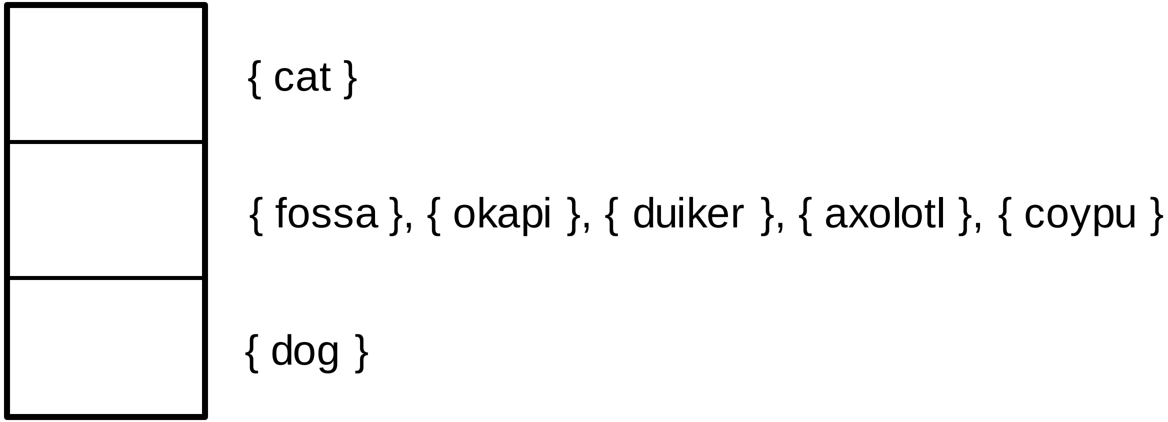
Flashback to Lecture 7 Slides…

Hash Table Design (from Last Time)

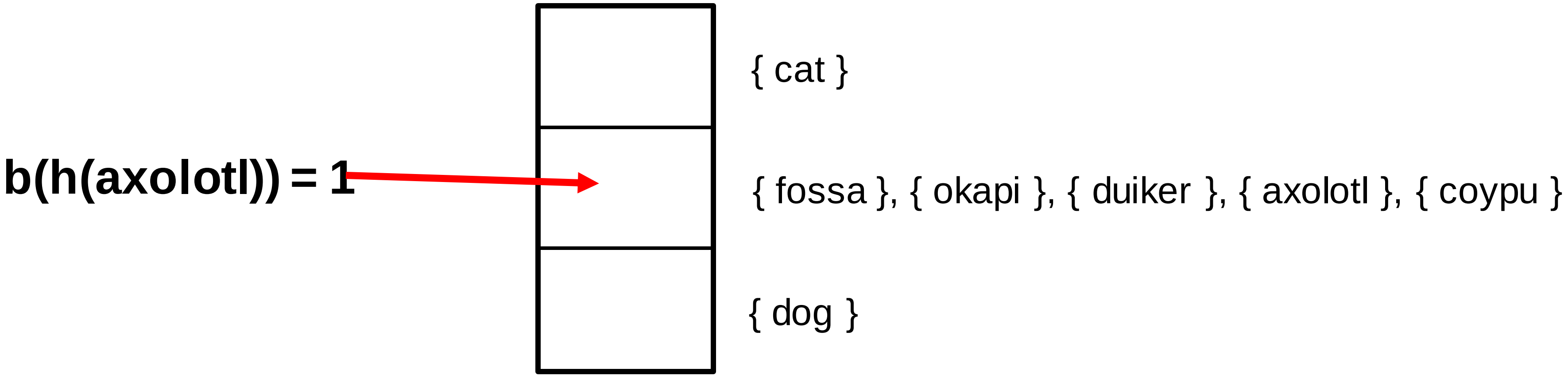
- Function b(c) maps hashcode c to bucket index j
- Every key with hashcode c goes into bucket b(c), in a linked list
- On find(k), must *walk the list* to find key matching k, if any

Hash Table with Chaining

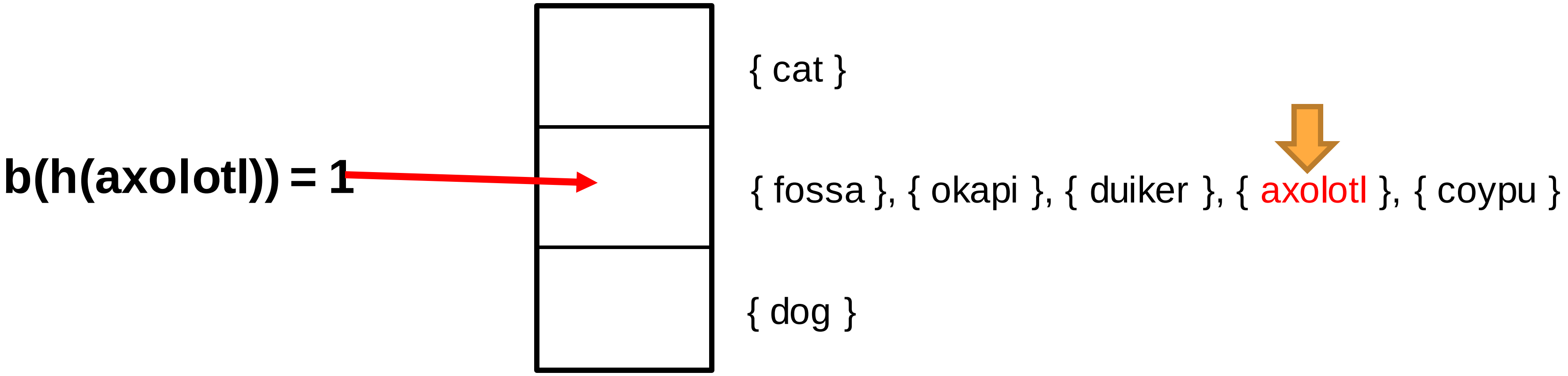
find(axolotl)



Hash Table with Chaining



Hash Table with Chaining



Hash Table Design (from Last Time)

- Function b(c) maps hashcodes c to bucket index j
- Every key with hashcode c goes into bucket b(c), in a linked list
- On find(k), must walk the list to find key matching k, if any
- **Quickie quiz: how do I compare key to each element of chain?**

Hash Table Design (from Last Time)

- Function b(c) maps hashcodes c to bucket index j
- Every key with hashcode c goes into bucket b(c), in a linked list
- On find(k), must walk the list to find key matching k, if any

- **Quickie quiz: how do I compare key to each element of chain?**
- **With equals() or similar – *not* with hashcodes! Why?**

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

$$\begin{array}{r} \color{red}{1234} \\ \times\, 0.6734 \\ \hline \end{array}$$

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
x0.6734
```

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
------------
```

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

Index Computation in Fixed-Point Decimal

- Suppose m = 100 = 10^2.

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

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,** $\left\lfloor \left((c \cdot A) \; mod \; 1.0 \right) \cdot m \right\rfloor = 97$

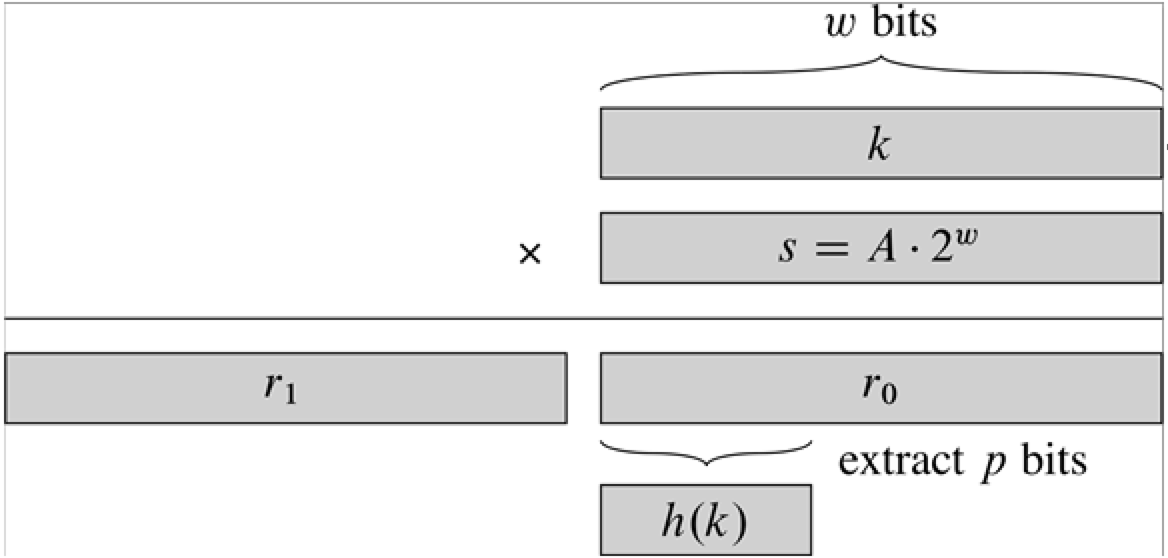
What About Fixed-Point Binary?

- Book presents the binary version.

- It's also how you would typically implement it on a computer!

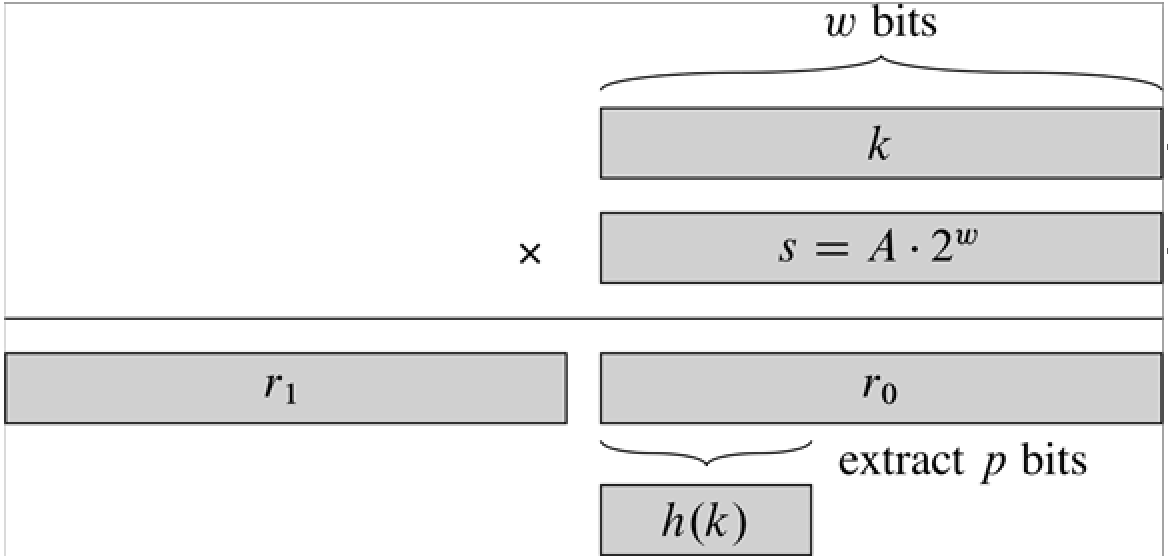
- If you have had 132, then the following slides will make more sense
  - If not, follow along as best you can, and look at this again after you've had 132

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



result of `.hashCode()`

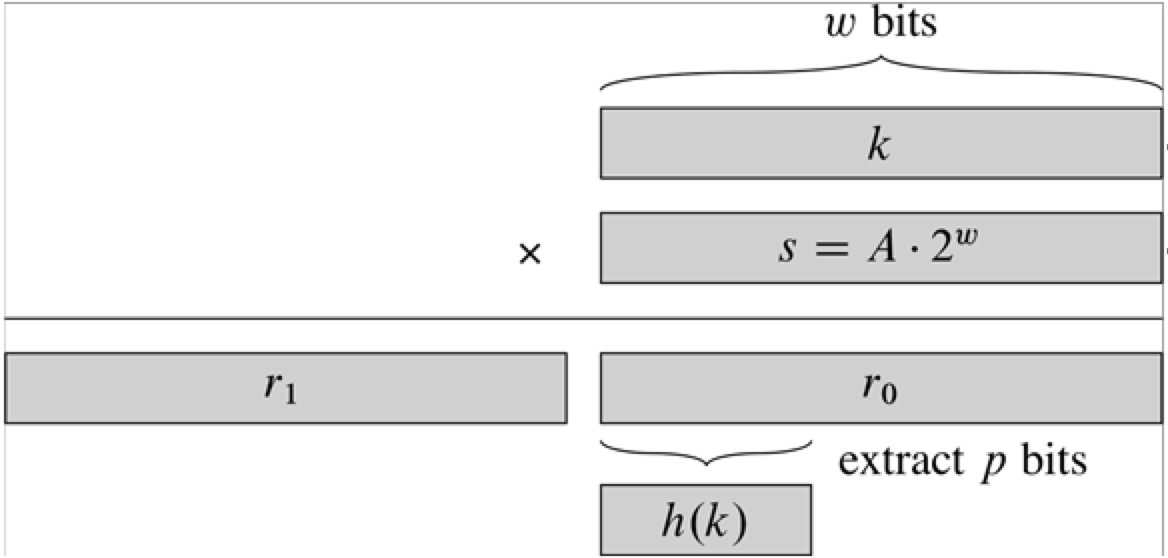
For base 2 (let's assume w = 32)



result of `.hashCode()`

A shifted left by 32 bits

For base 2 (let’s assume w = 32)

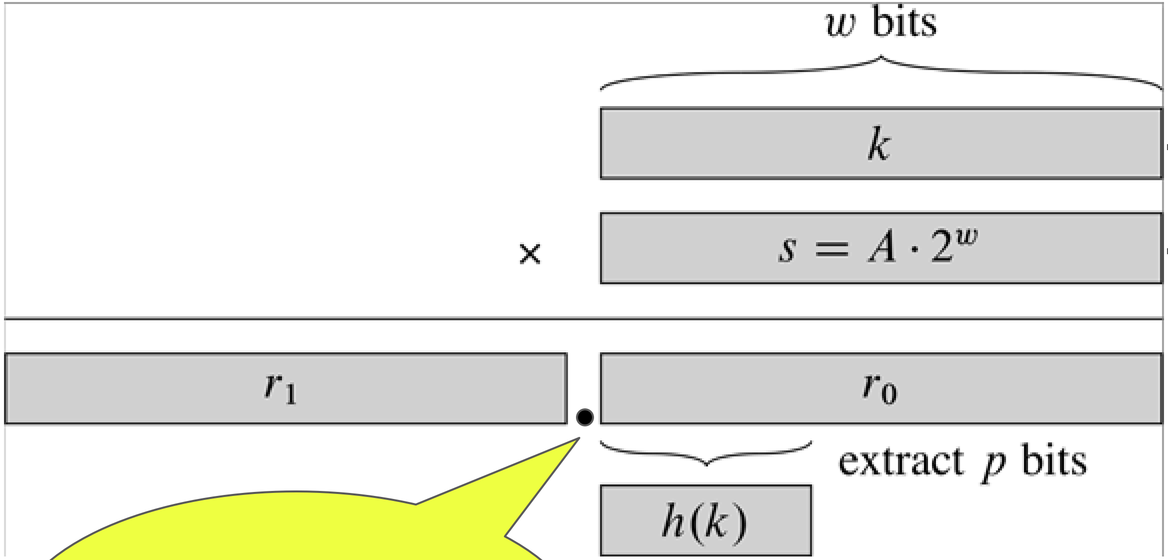


result of `.hashCode()`

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)

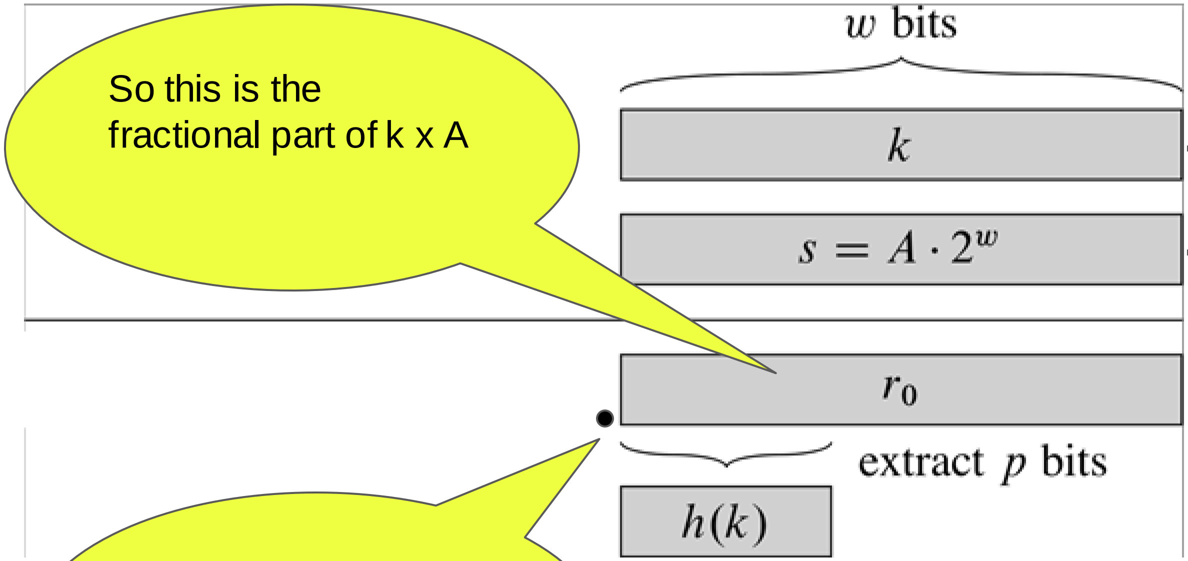


result of `.hashCode()`

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)



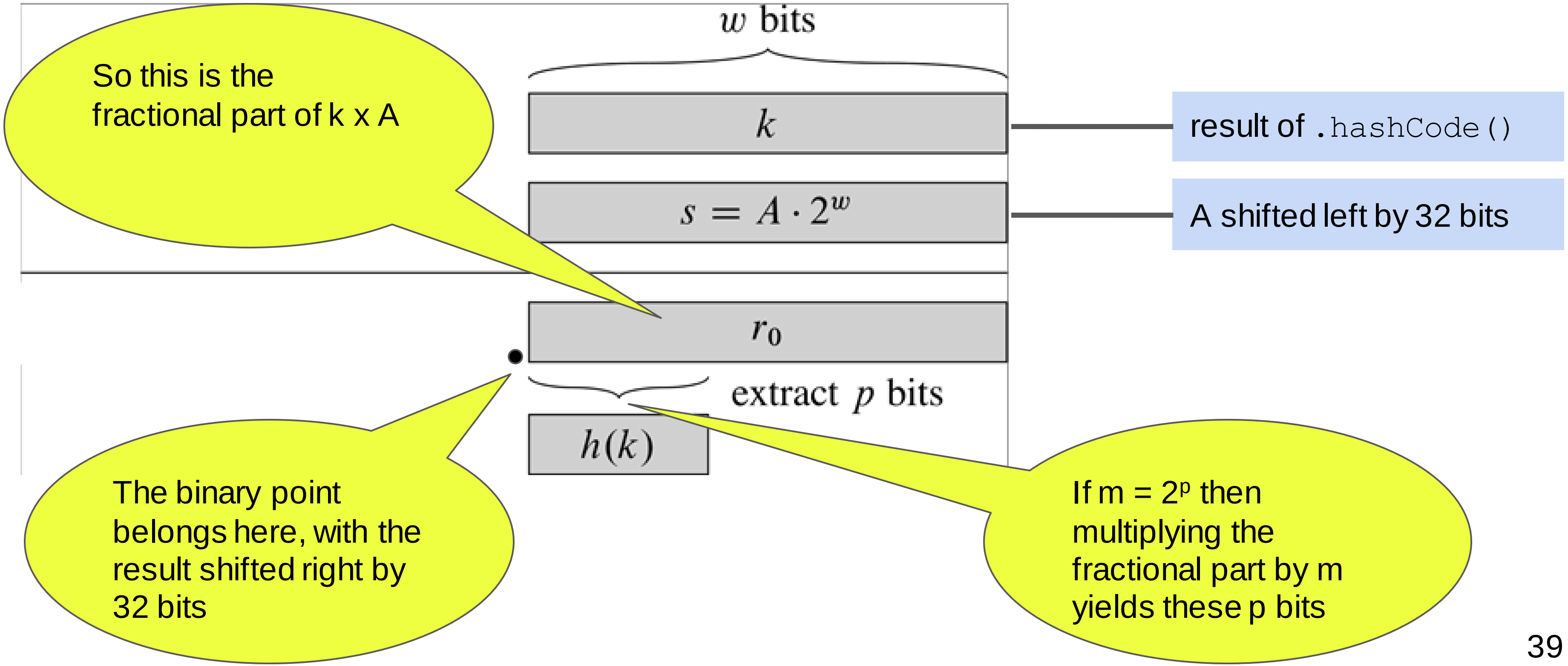
So this is the fractional part of k x A

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

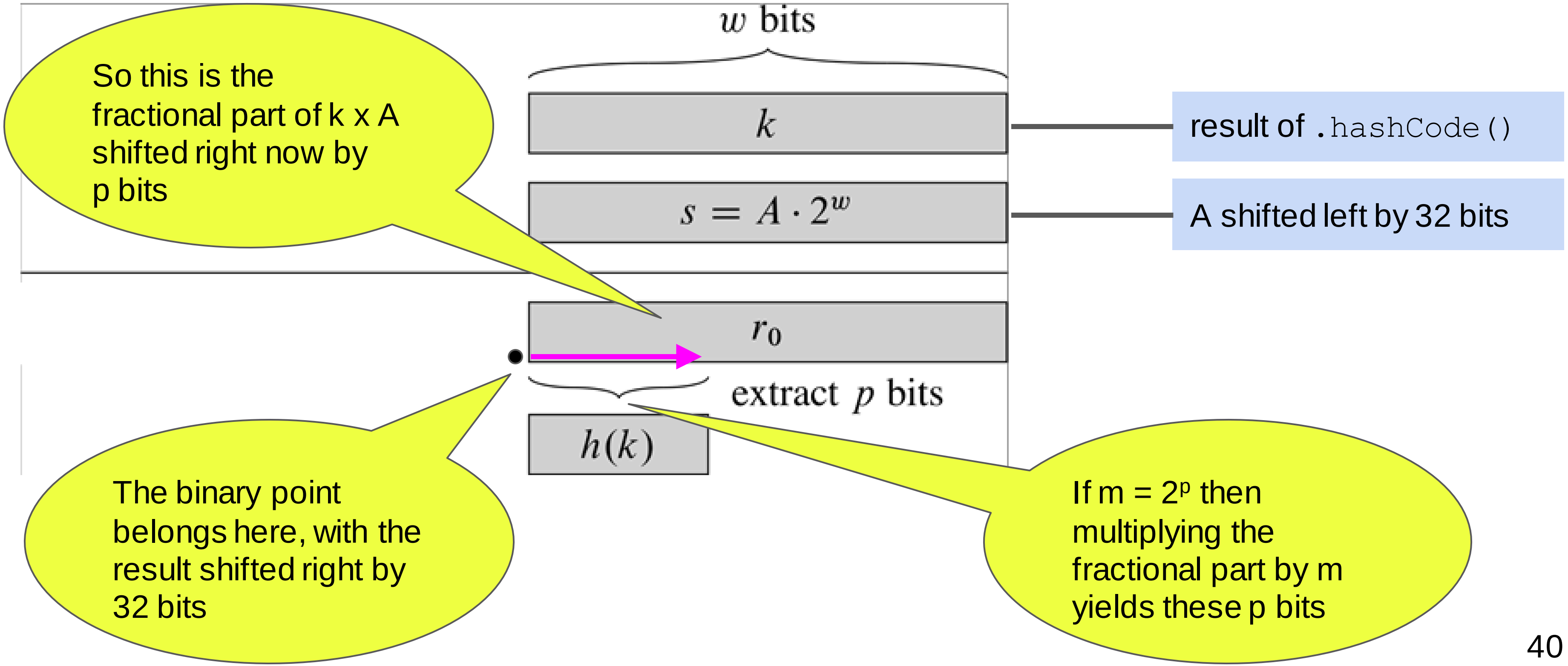
result of `.hashCode()`

A shifted left by 32 bits

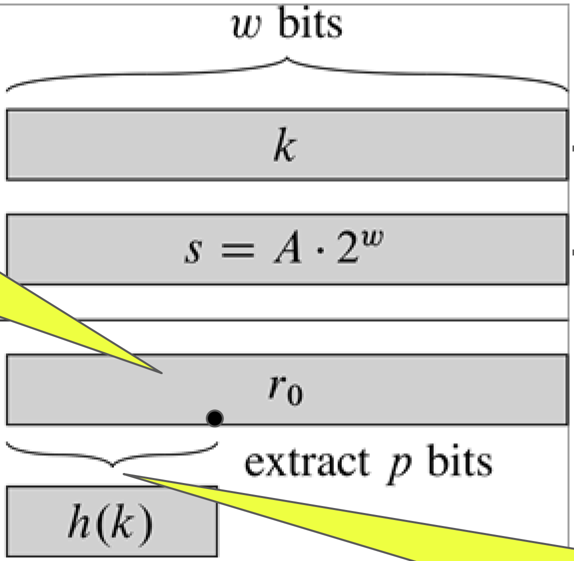
For base 2 (let's assume w = 32)



For base 2 (let’s assume w = 32)



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



result of `.hashCode()`

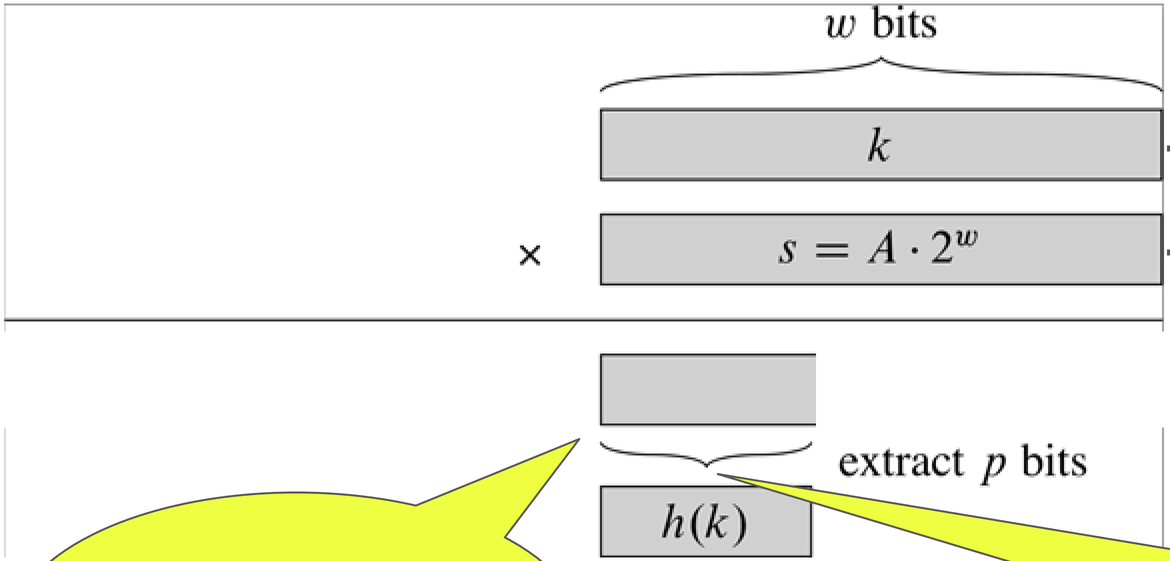
A shifted left by 32 bits

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)



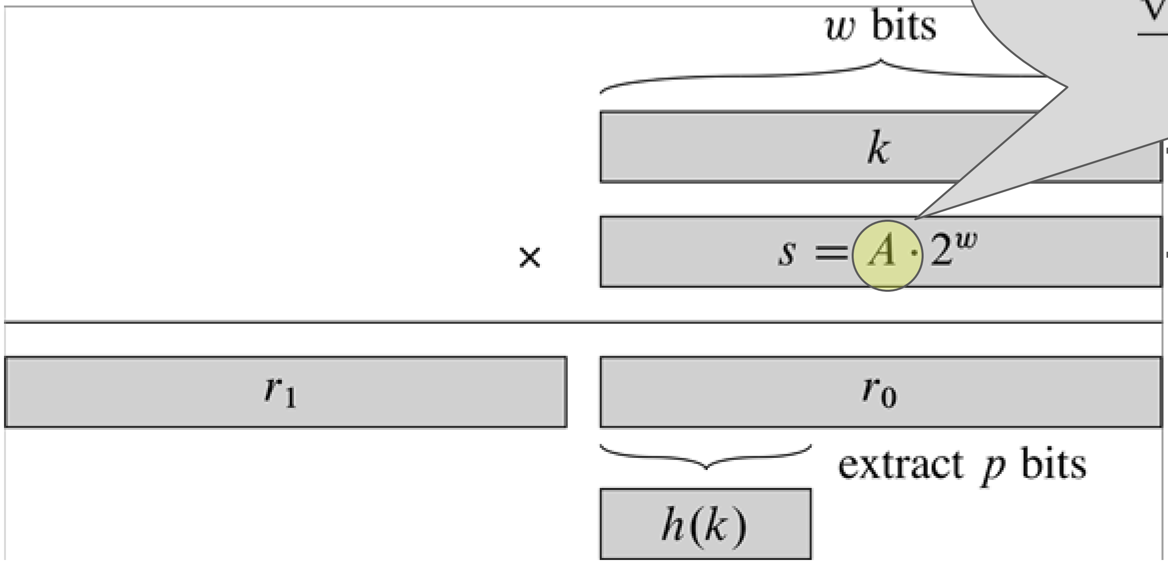
result of `.hashCode()`

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:

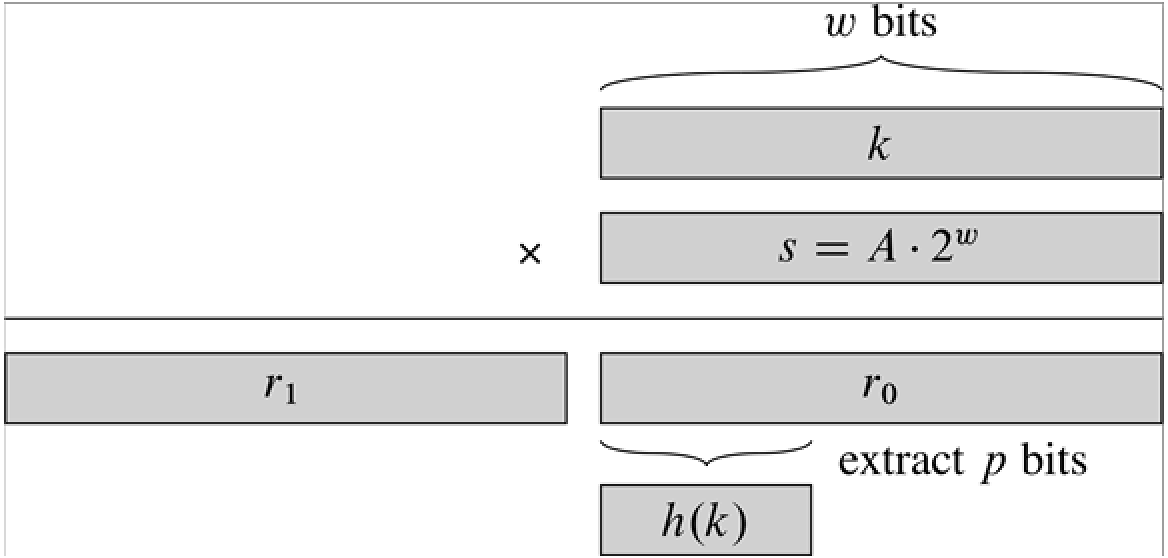
$$\frac{\sqrt{5}-1}{2} \approx 0.6180339887\ldots$$

result of `.hashCode()`

A shifted left by 32 bits

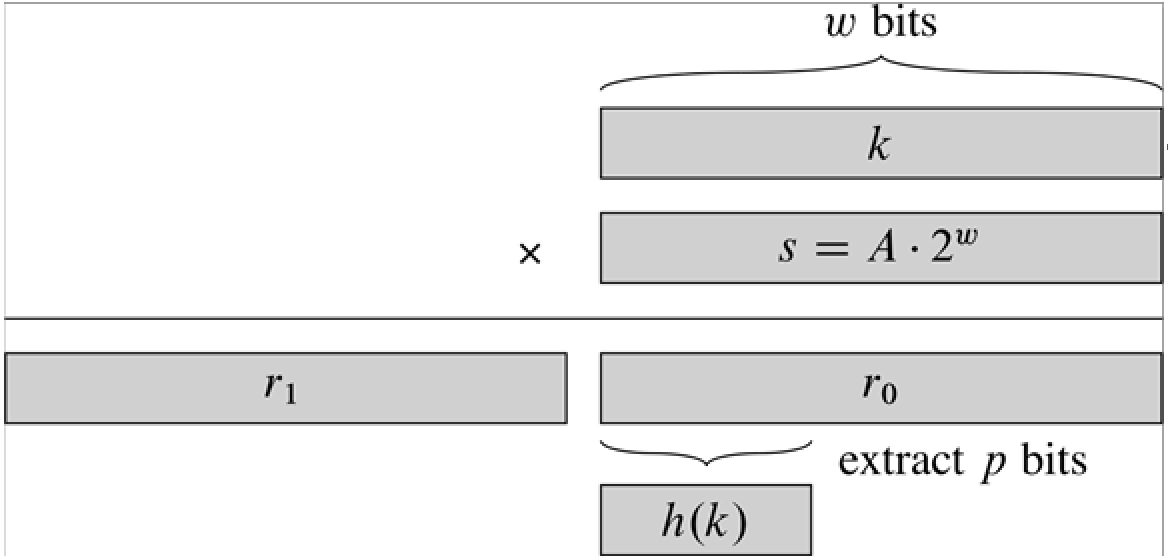
Example (page 264 in text)

w = 32
p = 14 → m = 16384



Example (page 264 in text)

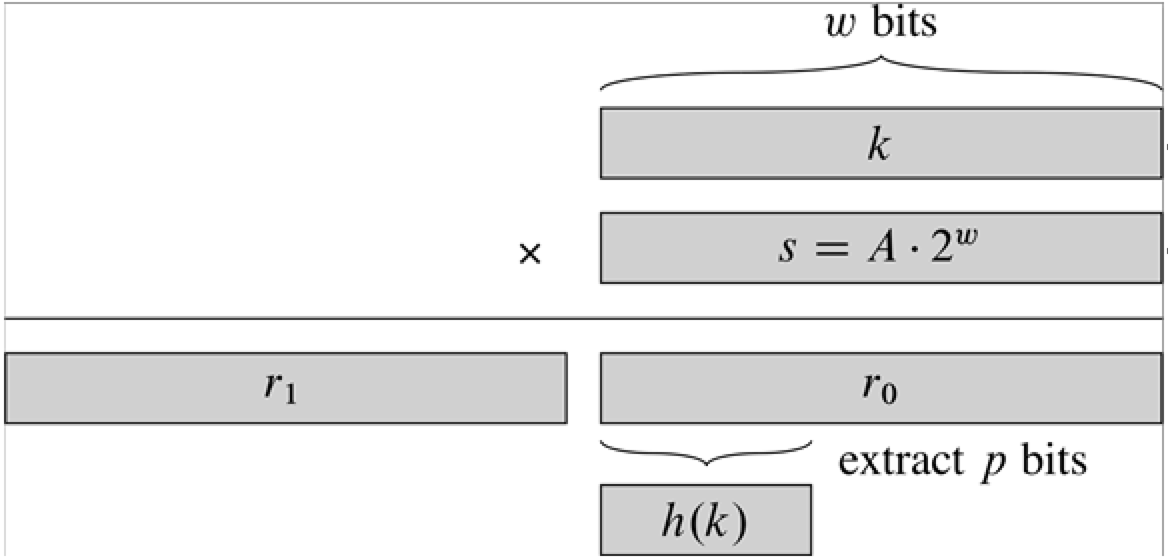
w = 32
p = 14 → m = 16384



k = 123456

Example (page 264 in text)

w = 32
p = 14 → m = 16384

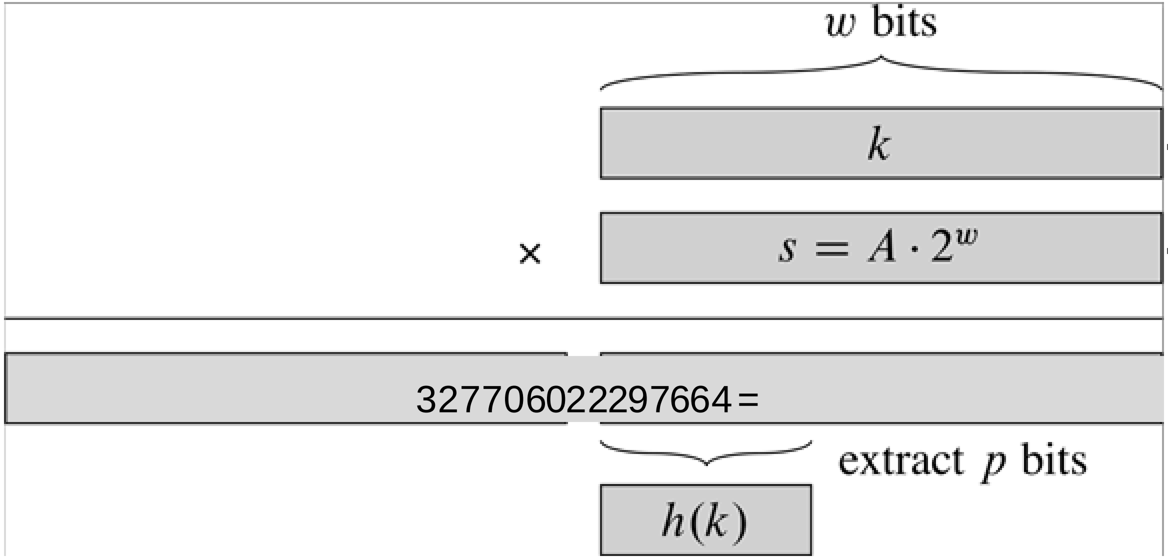


k = 123456

A x 2^{32} = 2654435769

Example (page 264 in text)

w = 32
p = 14 → m = 16384

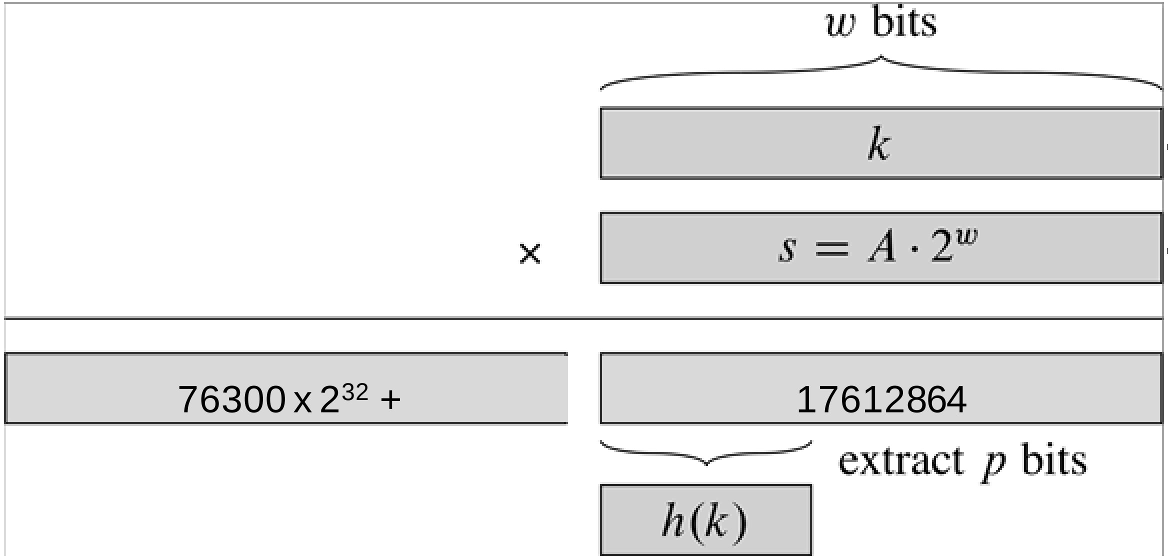


k = 123456

$A \times 2^{32} = 2654435769$

Example (page 264 in text)

w = 32
p = 14 → m = 16384

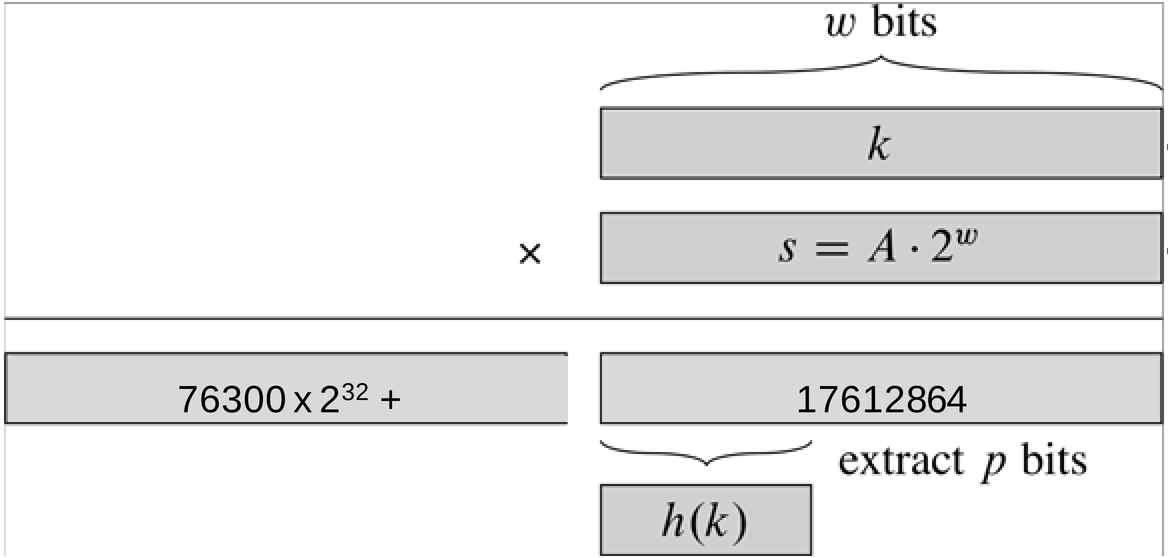


k = 123456

A x 2^{32} = 2654435769

Example (page 264 in text)

w = 32
p = 14 → m = 16384



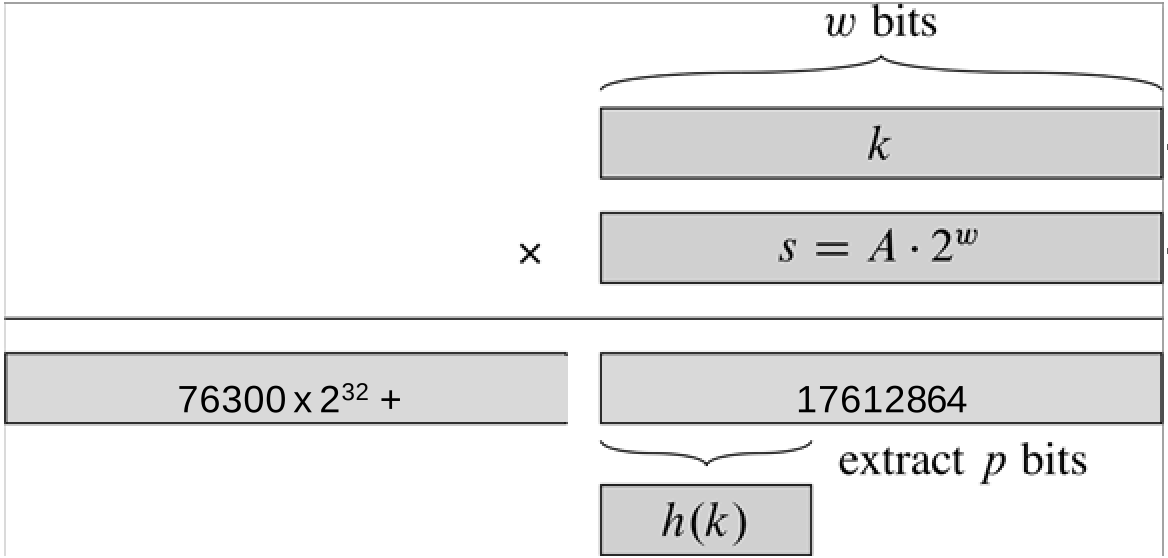
k = 123456

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

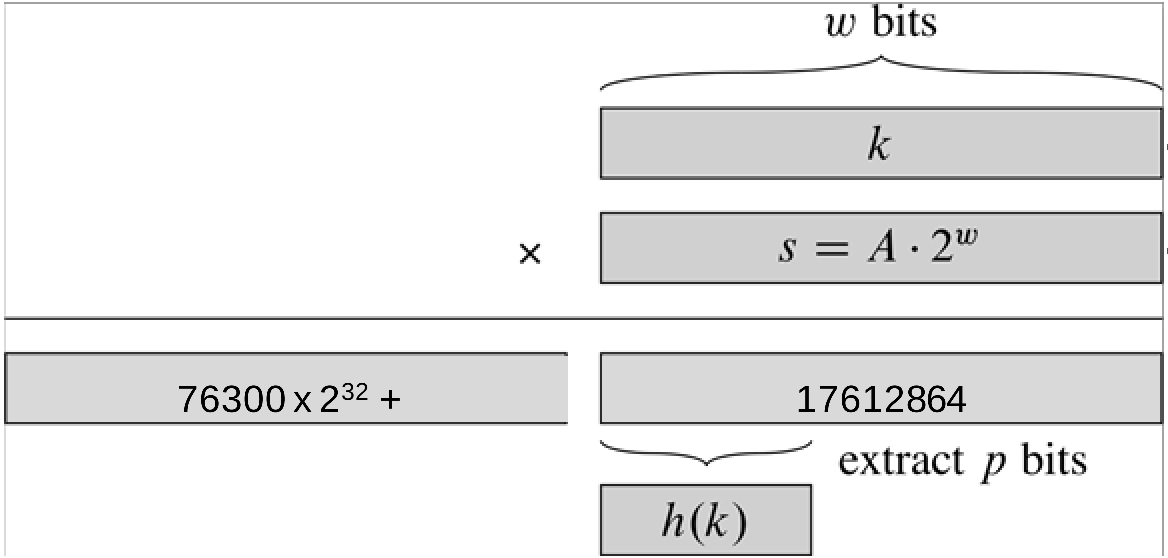
A x 2^{32} = 2654435769

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

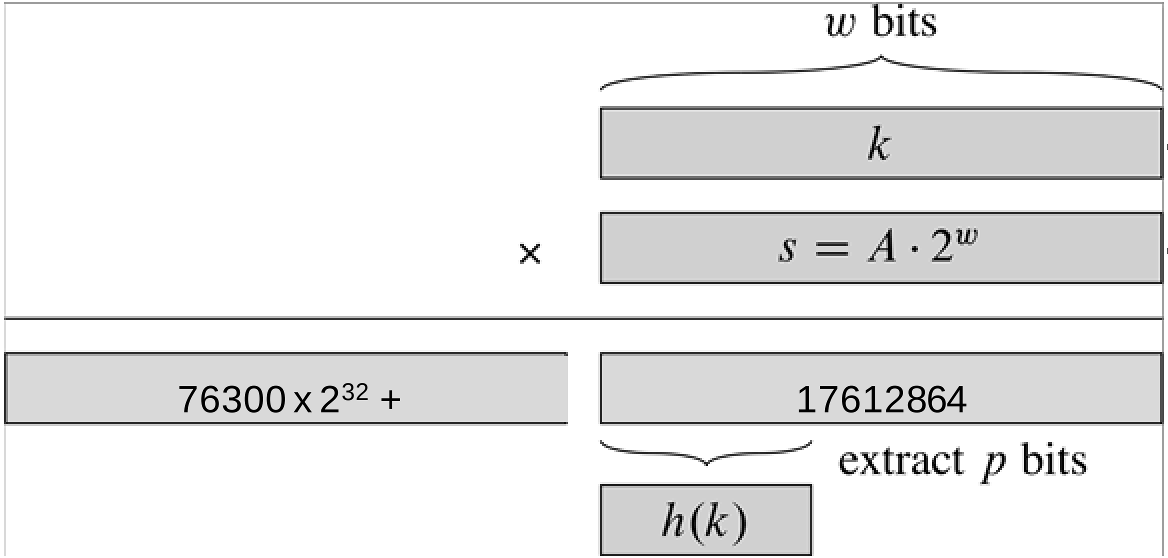
A x 2^{32} = 2654435769

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

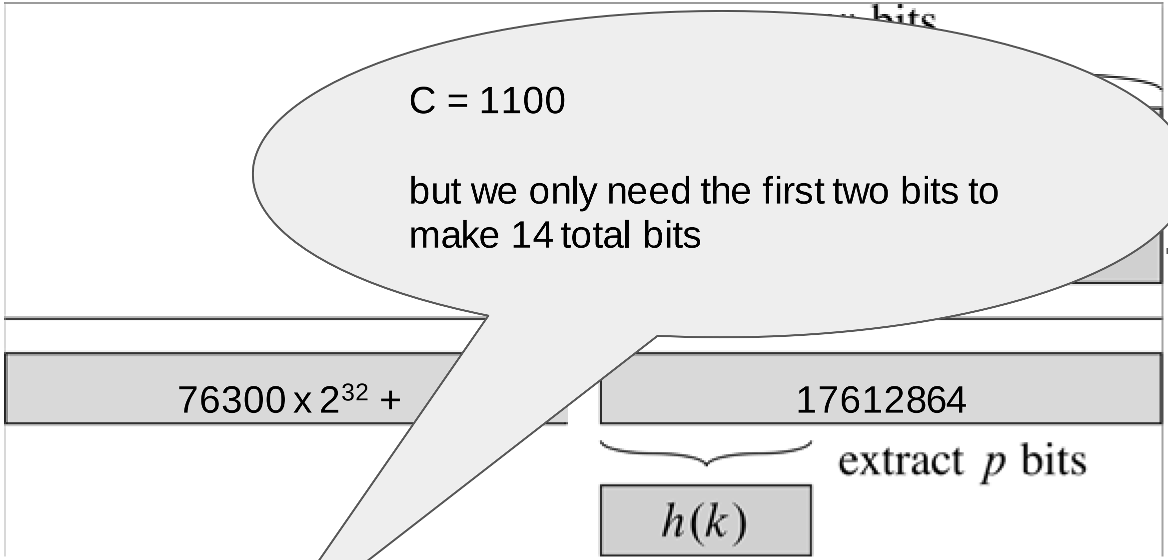
A x 2^{32} = 2654435769

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



C = 1100

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

k = 123456

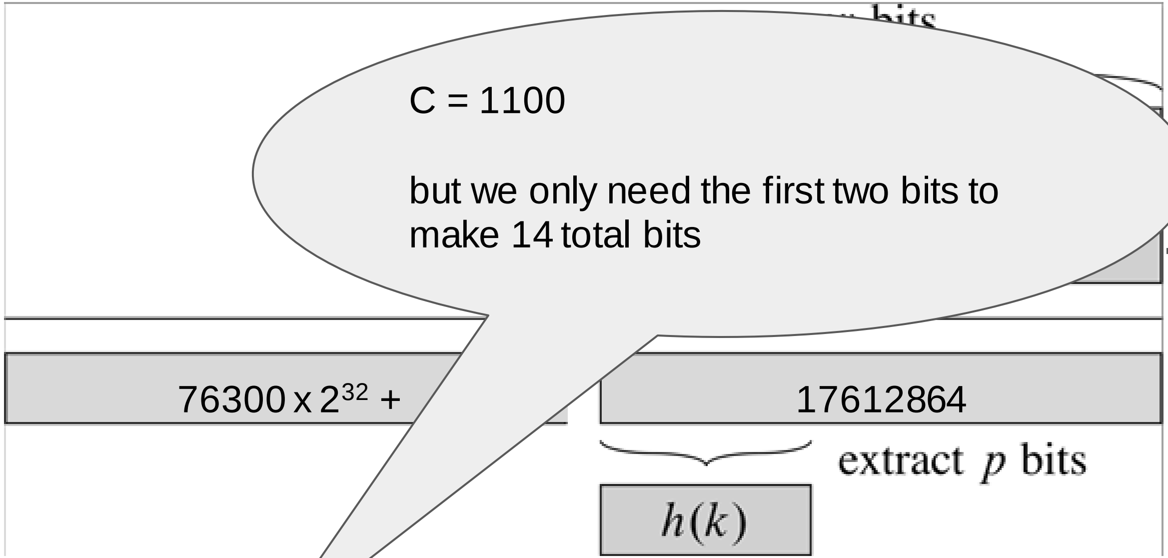
A x 2^{32} = 2654435769

Top 14 bits
01 0C
0000 0001 0000

32 bit representation for 17612864 is
01 0C C0 40

Example (page 264 in text)

$w = 32$
$p = 14 \rightarrow m = 16384$



C = 1100

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

$k = 123456$

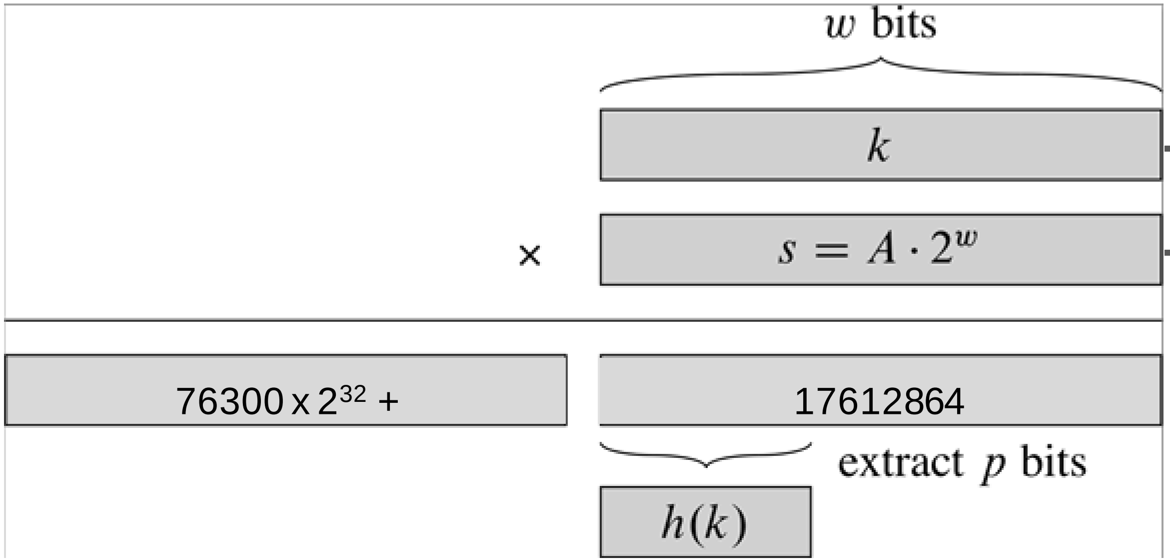
$A \times 2^{32} = 2654435769$

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 saw in Studio 7.*

New material

An Alternative Design – Open Addressing

- A chained hash table needs *two* data structures: arrays and lists

- Can we get by with just one data structure?
  - Simplicity is good
  - Lists can be slow

- **Open addressing**: hash tables, **unchained**

An Alte

- A chain d lists
- Can we
    - Sim
    - List
- **Open**

Idea: Open Addressing with Double Hashing

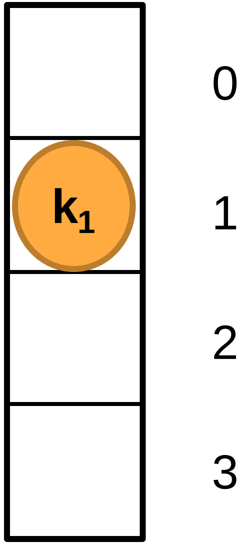
- Define **two** indexing functions: **b(c)** “base” and **s(c)** “step” that produce indices in [0,m)
- To **insert** record w/key hashcode c, first compute b(c) and s(c)
- Try to place record in table cell b(c)
- If that cell is full, try again at cell [b(c) + s(c)] mod m
- In general, try [b(c) = j*s(c)] mod m, j = 0,1,2,… until empty cell found

Open Addressing Example

- Suppose $m = 4$, $h(k_1) = c_1$, $b(c_1) = 1$, $s(c_1) = 3$

Open Addressing Example

- Suppose m = 4, $h(k_1) = c_1$, $b(c_1) = 1$, $s(c_1) = 3$

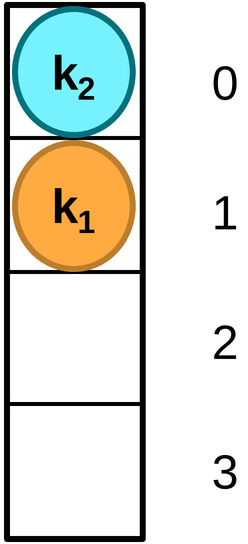


Open Addressing Example

- Suppose $m = 4$, $h(k_2) = c_2$, $b(c_2) = 0$, $s(c_2) = 1$

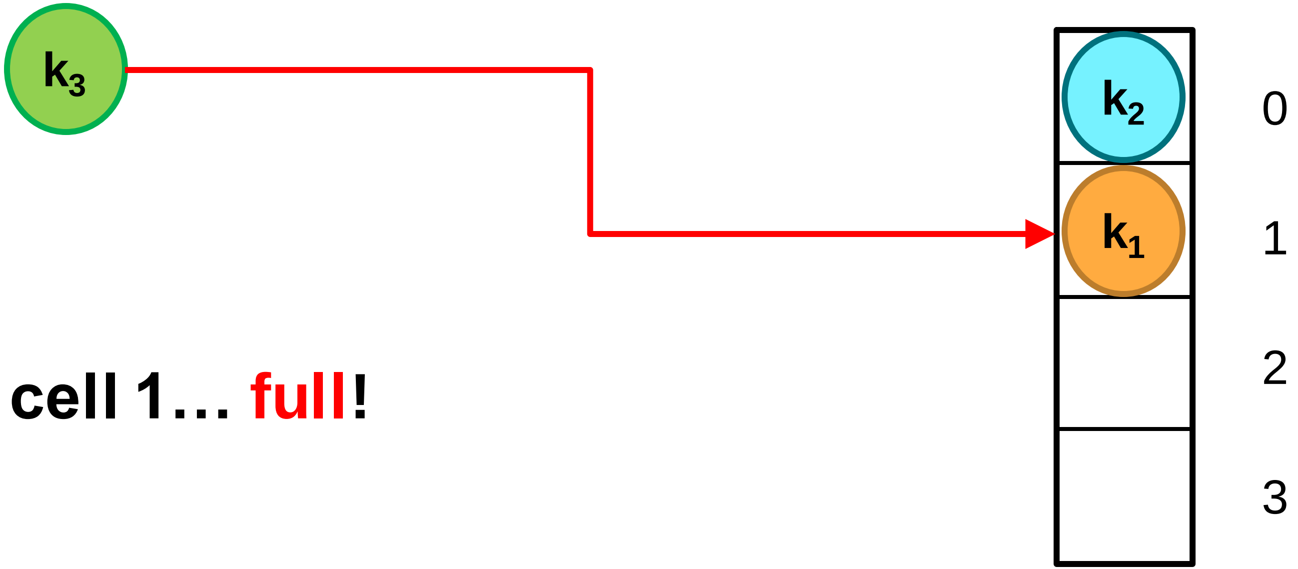
Open Addressing Example

- Suppose $m = 4$, $h(k_2) = c_2$, $b(c_2) = 0$, $s(c_2) = 1$



Open Addressing Example

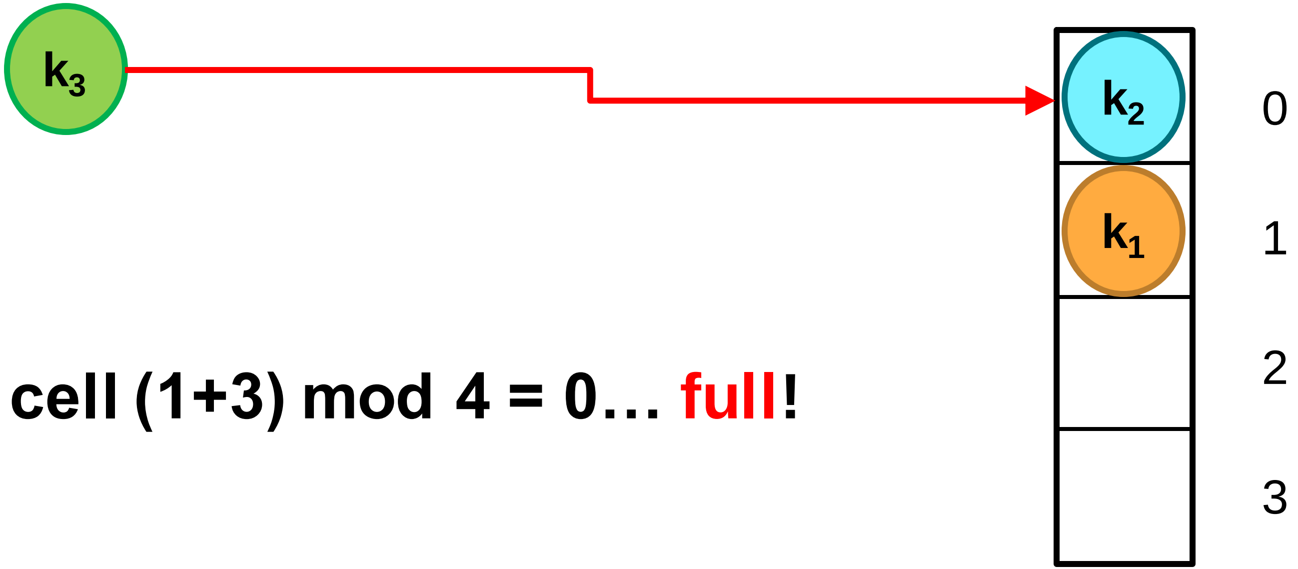
- Suppose $m = 4$, $h(k_3) = c_3$, $b(c_3) = 1$, $s(c_3) = 3$



Try cell 1… full!

Open Addressing Example

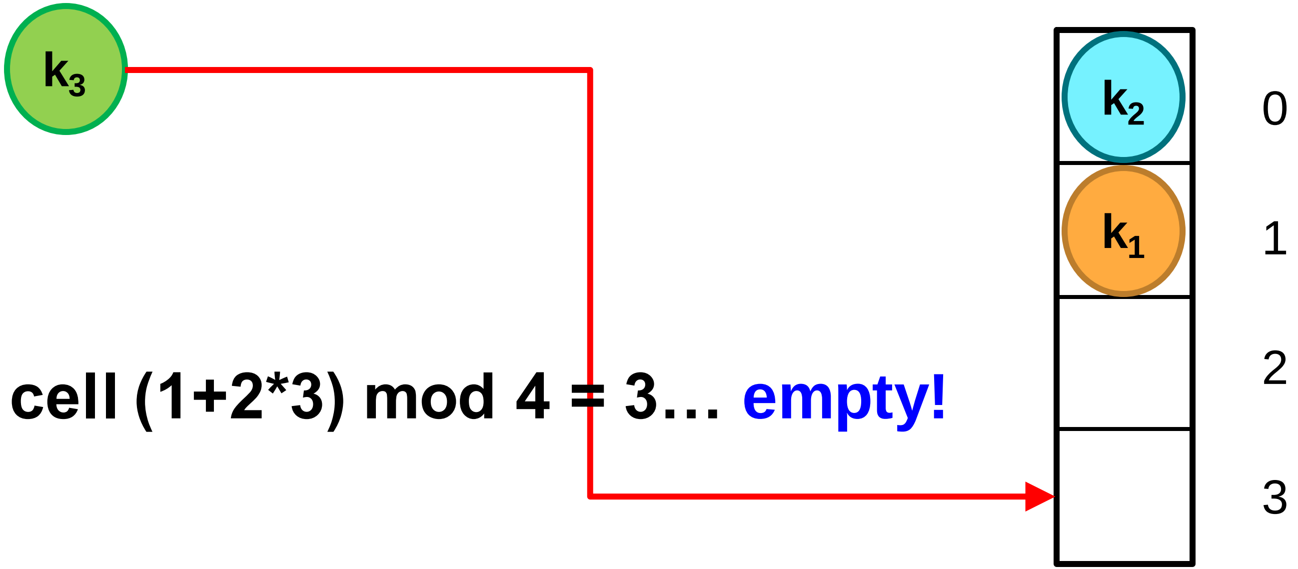
- Suppose m = 4, $h(k_3) = c_3$, $b(c_3) = 1$, $s(c_3) = 3$



Try cell (1+3) mod 4 = 0… full!

Open Addressing Example

- Suppose $m = 4$, $h(k_3) = c_3$, $b(c_3) = 1$, $s(c_3) = 3$

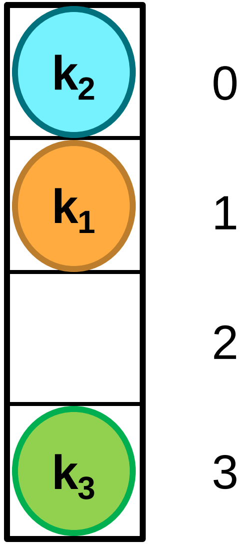


Try cell (1+2*3) mod 4 = 3… empty!

Open Addressing Example

- Suppose m = 4, $h(k_3) = c_3$, $b(c_3) = 1$, $s(c_3) = 3$

Try cell (1+2*3) mod 4 = 3



Notes on Open Addressing

- Find works similarly to insert – check cells as determined by b(c) and s(c) until desired key found (**success**), or an empty cell is found (**fail**)

- For correct operation:
  - Maintain load factor $\alpha<1$ (avg search time $\Theta(1/(1 - \alpha))$)
  - Make sure s(c) is *relatively prime* to m

    (e.g., s(c) always odd if m is power of 2) [why?]

Open Addressing: the Good

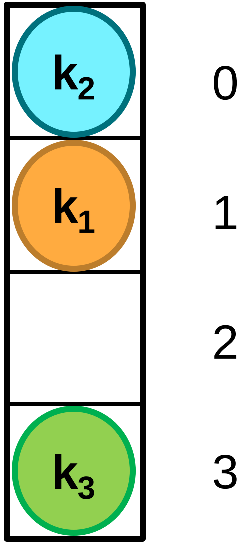
- Does not require linked lists (implicit in sequence of cells)
- Using two hash functions can resolve collisions faster
- If load factor $\leq 1/c$, $c > 1$, all ops still avg $\Theta(1)$ time

Open Addressing: the Bad

- Table can get full, unlike with chaining (resize!)
- Requires larger array for good performance w/given n
- Deletion is harder – cannot leave empty cells (why?)

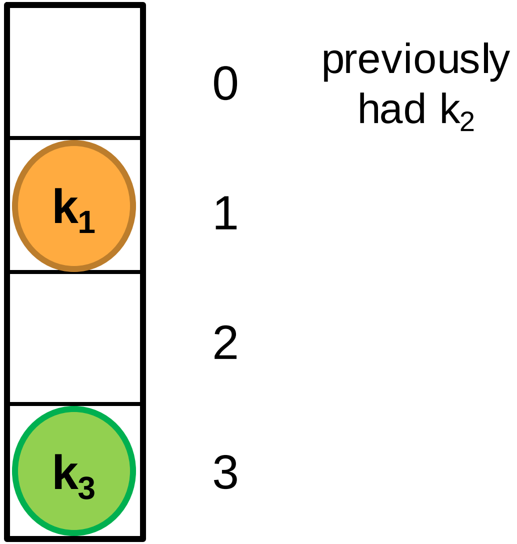
Open addressing – the Problem with Deletion

- remove(k_2)



Open addressing – the Problem with Deletion

- **remove(k_2)**

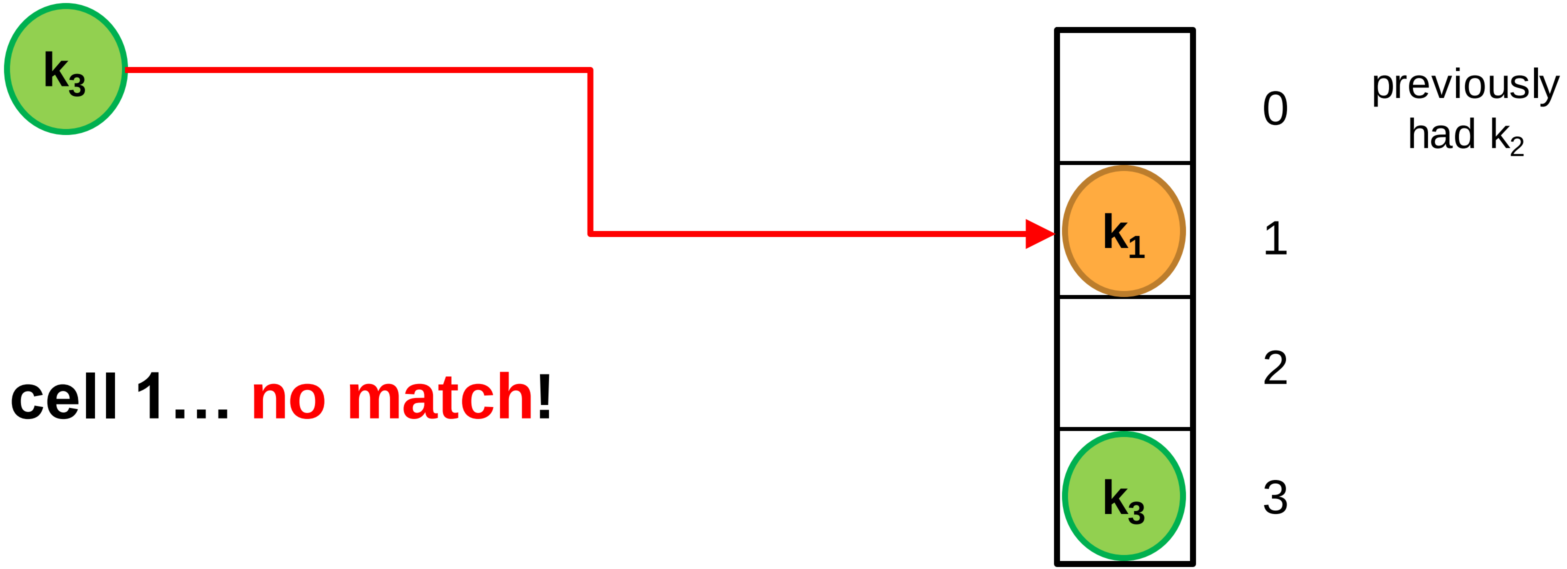


Open addressing – the Problem with Deletion

- find(k_3) → h(k_3) = c_3, b(c_3) = 1, s(c_3) = 3

Open addressing – the Problem with Deletion

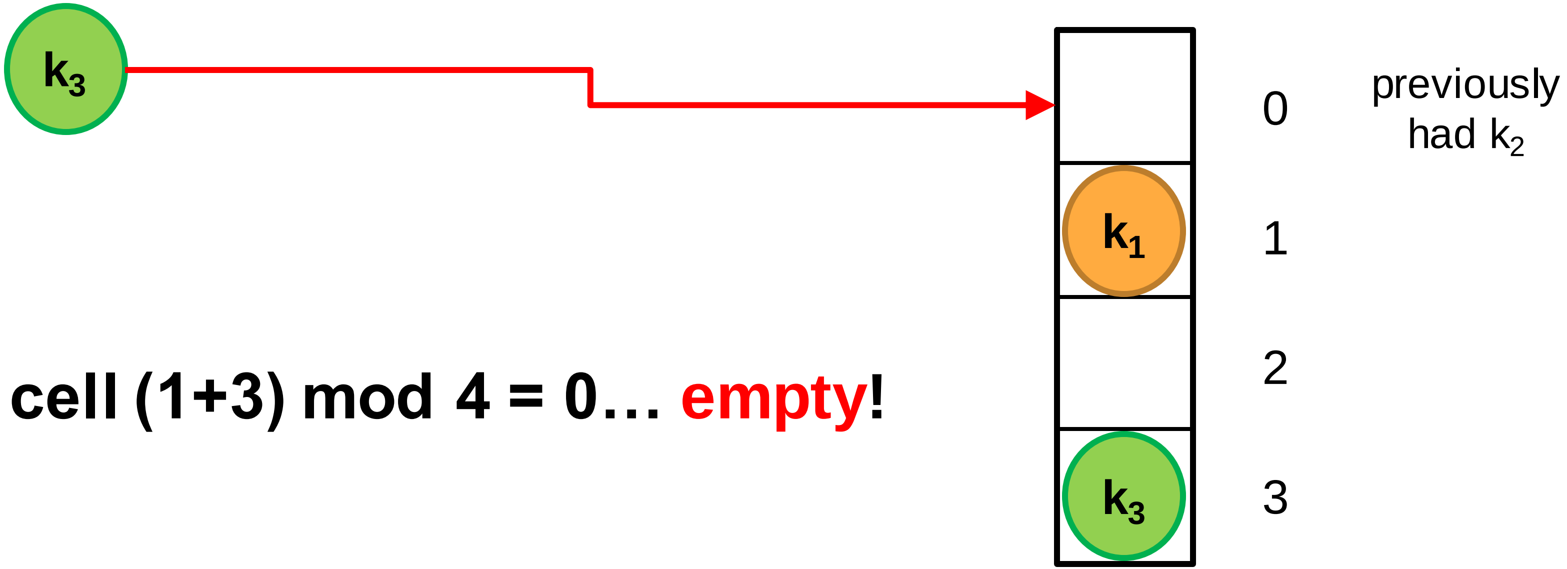
- find(k_3) → h(k_3) = c_3, b(c_3) = 1, s(c_3) = 3



Try cell 1… no match!

Open addressing – the Problem with Deletion

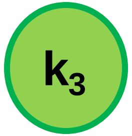
- find(k_3) → h(k_3) = c_3, b(c_3) = 1, s(c_3) = 3

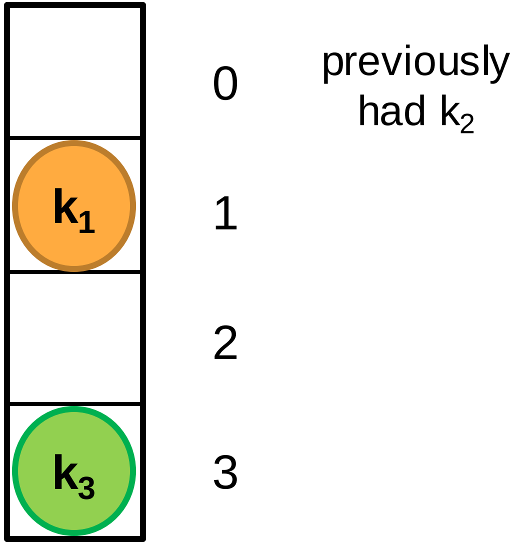


Try cell (1+3) mod 4 = 0… empty!

Open addressing – the Problem with Deletion

- find(k_3) → h(k_3) = c_3, b(c_3) = 1, s(c_3) = 3





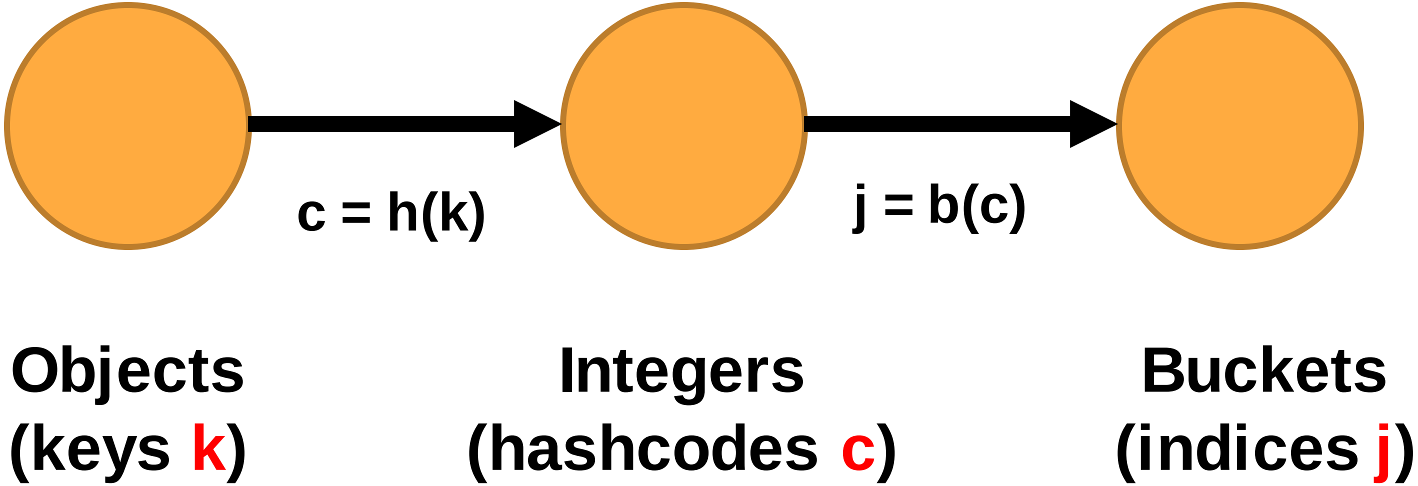
Returns "not found". Uh oh…

Open Addressing: the Bad

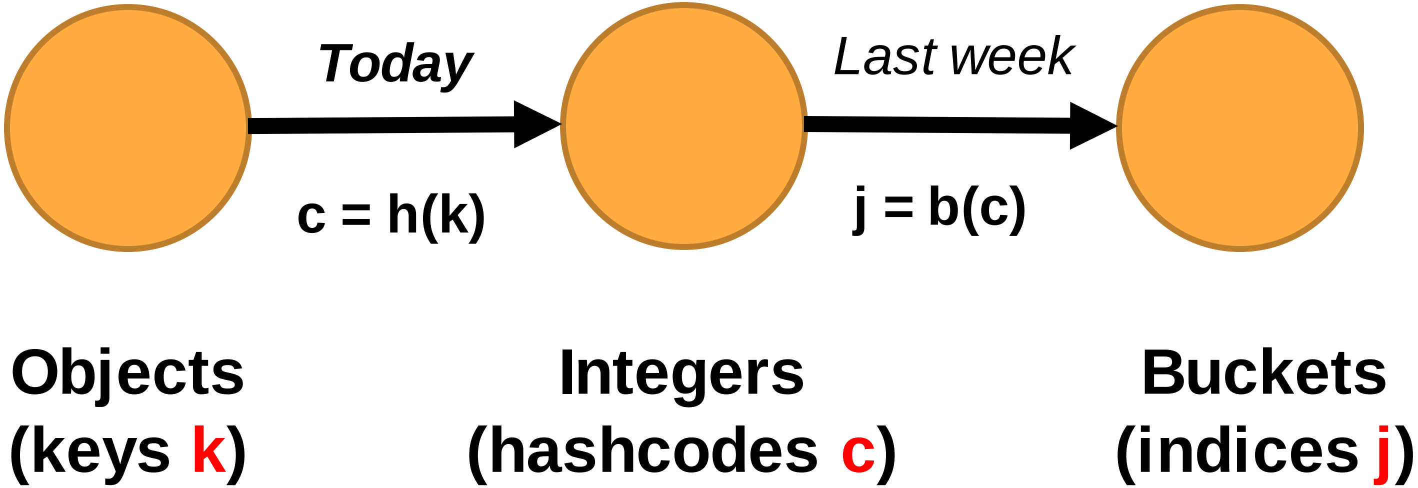
- Table can get full, unlike with chaining (resize!)
- Requires larger array for good performance w/given n
- Deletion is harder – cannot leave empty cells
- (*Deletion must leave behind a "deleted" marker so find does not stop prematurely*.)

And now, back to hash function design…

Hash Function Pipeline – Two Steps



Hash Function Pipeline – Two Steps



Purpose of Hashcode Generation

- Map objects to integers in some range
- Objects that are equal() must have **???** hashcode

Purpose of Hashcode Generation

- Map objects to integers in some range
- Objects that are equal() must have **the same** hashcode (Studio 7)
- Objects that are not equal() should have **???** hashcodes

Purpose of Hashcode Generation

- Map objects to integers in some range
- Objects that are equal() must have **the same** hashcode (Studio 7)
- Objects that are not equal() should have **distinct** hashcodes (but this may not always be possible due to PHP)
- **Question**: should hashcodes be spread uniformly across range without obvious correlations?

Argument About Hashcode Generation

- **Question**: should hashcodes be spread uniformly across range without obvious correlations?

- **No**: second step (index generation) is responsible for ensuring that unequal hashcodes are mapped to uniform, uncorrelated indices

- **Yes**: index generation is not responsible for “fixing” a bad hashcode generator

Argume

- **Ques** ... range
  withou

- **No**: se ... uring that
  unequ ... indices

- **Yes**: i ... hashcode
  gener

Different languages/code libraries take different sides in this argument.

Java implementation details (e.g. Color) suggest it thinks that "no, hashcodes need not be uniform/uncorrelated".

Some C++ implementations (e.g. MS VS 2015) expect uniformity; some don't.

Argu

- **Qu** ... ge
  wi

- **No** ... g that
  un ... ces

- **Ye** ... shcode
  ge

What assumption does your favorite language/library make?

And if it doesn't require uniform hashcodes, how good is its index generator?

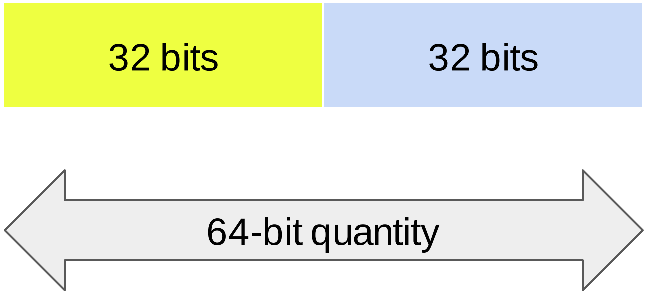
E.g. OpenJDK 8 HashMap:
table size is power of 2,
index = hashcode XOR hashcode/2^{16}

An Argument for “Good” Hashcodes, Regardless

- Even if your index generator scrambles the hashcode…
- … if universe of objects is much bigger than # possible hashcodes…
- … then *non-uniformity, correlations increase practical likelihood that you’ll encounter many objects that map to the* ***same*** *hashcode.*
- [This was not an issue in Studio 7, because # Color objects = # hashcodes]

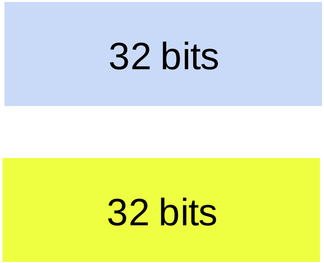
Hashcode Ideas for Primitive Types [from JDK8]

- (Arbitrary) 32-bit integers – use unchanged
- 32-bit floating point – use underlying bits as integer (*floatToIntBits()*)
- 64-bit long (including double-precision float via *doubleToLongBits()*) – hashcode = [value XOR (value / 2^{32})] mod 2^{32}



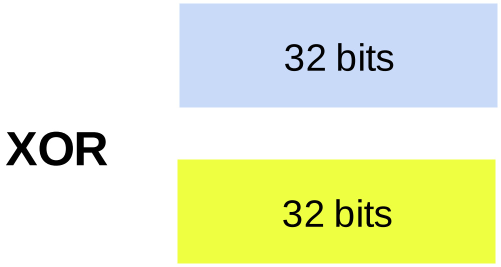
Hashcode Ideas for Primitive Types [from JDK8]

- (Arbitrary) 32-bit integers – use unchanged
- 32-bit floating point – use underlying bits as integer (*floatToIntBits()*)
- 64-bit long (including double-precision float via *doubleToLongBits()*) – hashcode = [value XOR (value / 2^{32})] mod 2^{32}



Hashcode Ideas for Primitive Types [from JDK8]

- (Arbitrary) 32-bit integers – use unchanged
- 32-bit floating point – use underlying bits as integer (*floatToIntBits()*)
- 64-bit long (including double-precision float via *doubleToLongBits()*) – hashcode = [value XOR (value / 2^{32})] mod 2^{32}



Hashcode Ideas for Primitive Types [from JDK8]

- (Arbitrary) 32-bit integers – use unchanged
- 32-bit floating point – use underlying bits as integer (*floatToIntBits()*)
- 64-bit long (including double-precision float via *doubleToLongBits()*) – hashcode = [value XOR (value / 2^{32})] mod 2^{32}

32 bits

Hashcode Ideas for Primitive Types [from JDK8]

- (Arbitrary) 32-bit integers – use unchanged
- 32-bit floatin … (*floatToIntBits()*)
- 64-bit long (i … *leToLongBits()*) –
  hashcode = …

If you also want uniformity/decorrelation, use e.g. multiplicative hashing strategy with $m=2^{32}$ to map these values to hashcodes.

32 bits

More Hashcode Ideas for Primitive Types

- Types with limited # of values (e.g. Booleans, enums)?

- Cannot hope to cover entire space of hashcodes

- Either map to small ints & rely on index calc to scramble…

- Or guess a mapping with “nice” properties
- E.g. Boolean: true → 1231, false → 1237 (*why?*)

Hashing Composite Objects

- More complex datatypes come in two flavors

- **Sets** – collection of objects, **no order** [e.g. Java Set]:
- {3, 2, 5} = {2, 5, 3} = {5, 3, 2}

- **Sequences** – collection of objects, **order matters** [e.g. List, String]:
- [2, 5, 3] ≠ [3, 2, 5]

- “k-tuple” – sequence of k objects [$o_1 \ldots o_k$] of types [$t_1 \ldots t_k$]
- (objects of class type with data members)

Hashing Sets and Sequences

- Assume we have hashcodes for each element in composite object
- How do we construct a *single* hashcode for the whole object?
- **Sets** – must get same result regardless of element order
- E.g., $h(c_1 \ldots c_k)$ = **???**

Hashing Sets and Sequences

- Assume we have hashcodes for each element in composite object
- How do we construct a *single* hashcode for the whole object?
- **Sets** – must get *same result* regardless of element order
- E.g., $h(c_1 \ldots c_k) = \sum_j c_j$ **or** $h(c_1 \ldots c_k) = \min_j c_j$
- **Sequences** – hashcode may (should!) depend on element order

Aside: Strings are a Kind of Sequence

- Sequence of characters (8- or 16-bit values)

- Must compare using equals() [contents same], **not** == [memory same]
  - “if key == record.key” might return false when strings are equal!!!
  - Instead, say “if key.equals(record.key)”

How Should We Hash a Sequence?

- Need to combine multiple, perhaps variable #, of hashcodes into one
- Order should influence final hashcode
- Example (Java JDK 8, 10):

```
c ← 0
For each elt oj in sequence w/code cj
   c ← c * 31 + cj
```

Do We Like This Function? Why 31?

```
c ← 0
For each elt o_j in sequence w/code c_j
    c ← c * 31 + c_j
```

- **31 is prime →** does not just shift bits of c upward (better diffusion)
- **31 is 2^5 – 1 →** can avoid multiply because "x*31" is same as "(x << 5) – x" (faster on some processors)
- **31 is small →** can add more small hashcodes (e.g. characters) without overflowing and perhaps losing information

Do We Like This Function? Why 31?

```
c ← 0
For each elt oj in sequence w/code cj
    c ← c * 31 + cj
```

- But… it's easy to find many short sequences that map to same hashcode!
- *Why might this matter?*
- Probably should not rely on this fcn alone for decorrelation.

Example Alternative: Fowler-Noll-Vo Hashing

```
c ← 2166136261
For each elt oj in sequence w/code cj
    c ← (c XOR cj)*16777619
```

- Similar in spirit, but designed to scramble correlations in input
- $16777619 = 2^{24} + 2^{8} + 147$, so still pretty fast to multiply
- Original work assumes each c_j is one byte, e.g. English strings ($o_j = c_j$)
- [MANY other strategies to hash sequences can be found online]

Philosophical Musings

- Hashcode computation trades off efficiency vs scrambling
- How paranoid are you about input uniformity and correlations?
- (*In Studio 9, we'll be extra-paranoid – malicious adversary*)
- Ultimately, must test hash fcns empirically, assess risks vs benefits
- Language/library defaults aren't always what you'd like.

End of Lecture 9