Hash Function

TODAY

- What Makes a Good Hash Function?

- Basic Hash Functions

What Makes a Good Hash Function?

- Efficiency of Computation
- Minimizing Collisions

Efficiency of Computation

- Integer vs Floating-Point Operations

- Arithmetic and Bitwise Operations

Integer vs Floating-Point Operations

- Integer
  - int, unsigned int, long long, unsigned long long
- Floating-Point
  - float, double

Integer vs Floating-Point Operations

Operation	Integer Latency	Integer Issue	Integer Capacity	Floating point Latency	Floating point Issue	Floating point Capacity
Addition	1	1	4	3	1	1
Multiplication	3	1	1	5	1	2
Division	3–30	3–30	1	3–15	3–15	1

Figure 5.12 **Latency, issue time, and capacity characteristics of reference machine operations.** Latency indicates the total number of clock cycles required to perform the actual operations, while issue time indicates the minimum number of cycles between two independent operations. The capacity indicates how many of these operations can be issued simultaneously. The times for division depend on the data values.

Integer vs Floating-Point Operations

- **Integer**
  - **Let's assume positive numbers**

ex) 12 = (1 * 2^3) + (1 * 2^2)

31	30	29	28	27	26		5	4	3	2	1	0
0	0	0	0	0	0	…	0	0	1	1	0	0
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}		2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

Integer vs Floating-Point Operations

- **Floating-Point**
  - Let's assume positive numbers

ex) 12.375 = (1 * 2^3) + (1 * 2^2) + (1 * 2^-2) + (1 * 2^-3)

=> 2^3 * {(1 * 2^0) + (1 * 2^-1) + (1 * 2^-5) + (1 * 2^-6)}

3 + 127 = 130

10000010

1.100011

100011

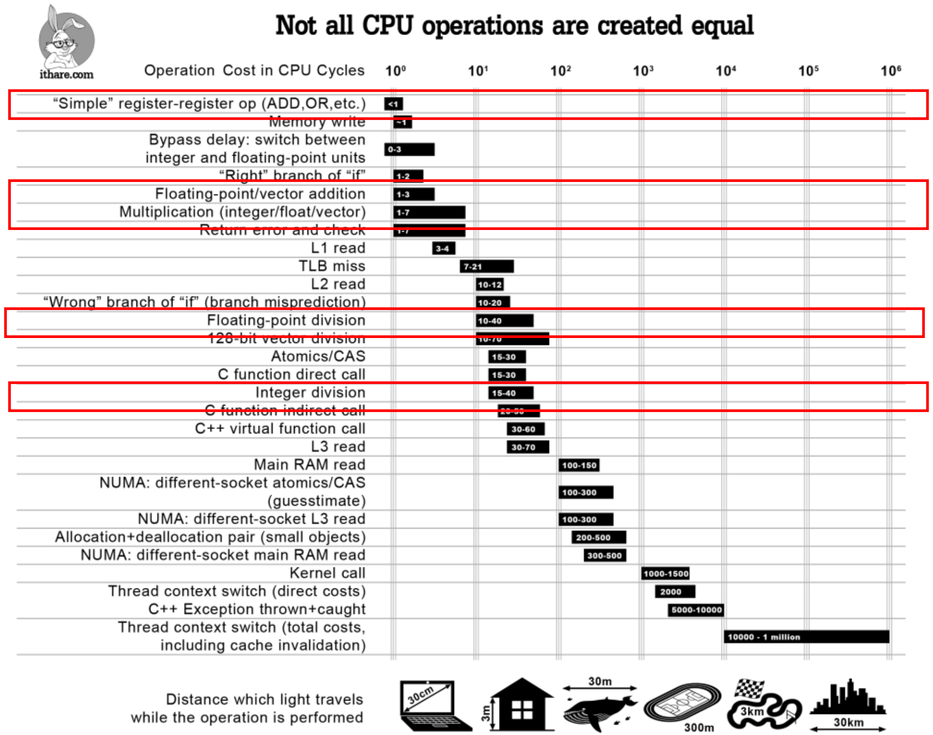
Sign (1bit)	Exponent (8bit)	Mantissa (23bit)

Efficiency of Computation

- Integer vs. Floating-Point Operations

- Arithmetic and Bitwise Operations

Arithmetic and Bitwise Operations



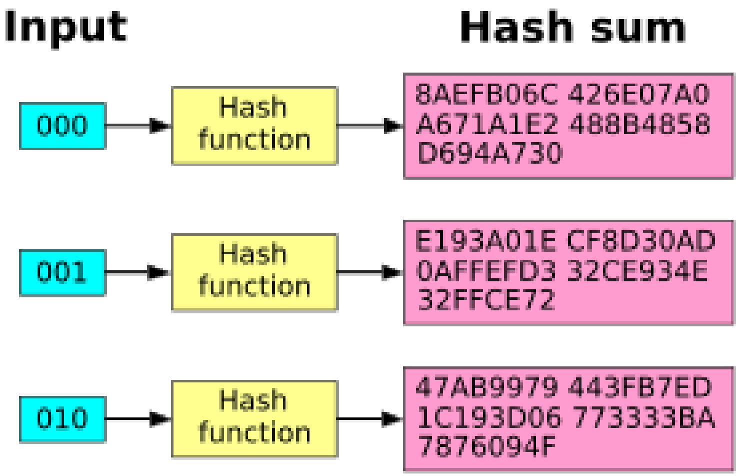
What Makes a Good Hash Function?

- Efficiency of Computation

- Minimizing Collisions

Minimizing Collisions

- Random Distribution of Hash Values
  - Avalanche Effect



TODAY

- What Makes a Good Hash Function?

- **Basic Hash Functions**

Basic Hash Functions

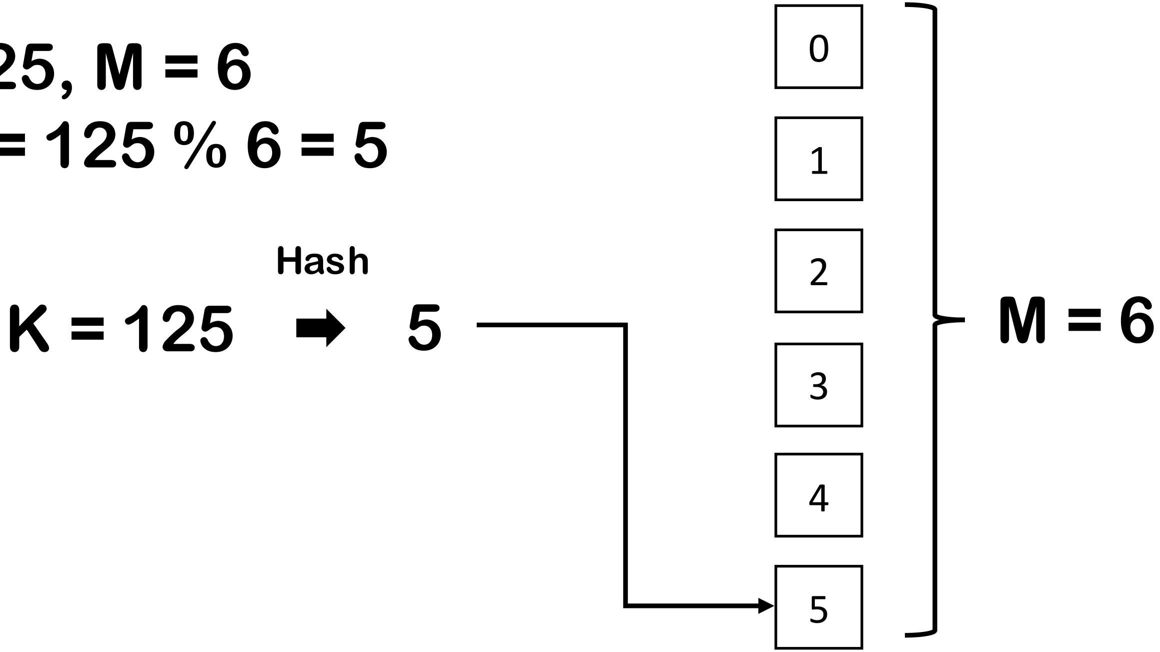
- Division Hashing

- Multiplicative Hashing

Division Hashing

index = K % M

ex) K = 125, M = 6
=> index = 125 % 6 = 5



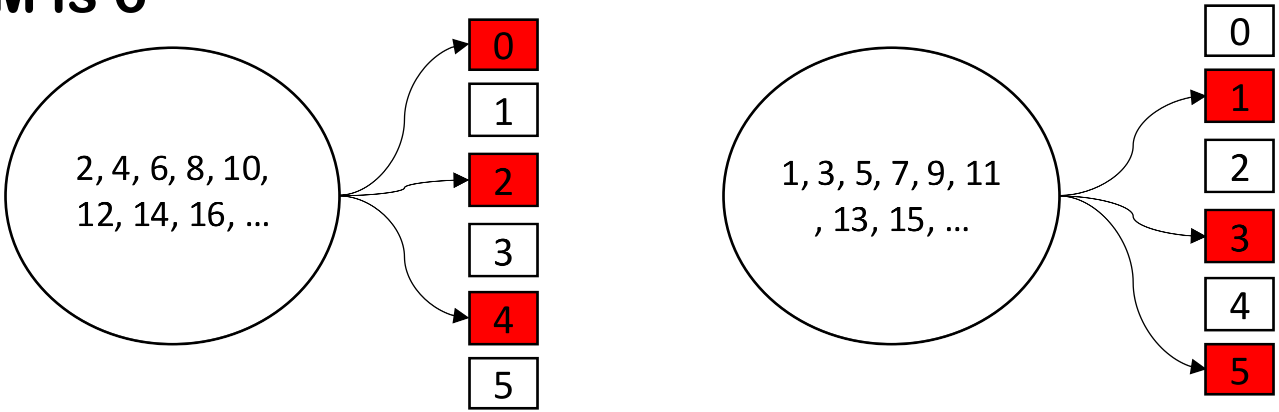
Impact of Bin Count (M)

- Multiple of 2
- Power of 2
- Composite Number
- Prime Number

Impact of Bin Count (M)

- **Multiple of 2**
  - **Even keys map to even bins**
  - **Odd keys map to odd bins**

ex) M is 6



Impact of Bin Count (M)

- Power of 2
  - Similar to multiples of 2, bins can become biased
  - But the modulo operation (%) can be replaced by the bitwise AND operation (&)

ex) K = 13, M = 2^2

=> K % M = 13 % 2^2 = 13 & (2^2 – 1)

Impact of Bin Count (M)

- **Power of 2**

ex) K = 13, M = 2^2

=> 13 = (1 * 2^3) + (1 * 2^2) + (1 * 2^0)

31	30	29	28	27	26
0	0	0	0	0	0
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}

…

5	4	3	2	1	0
0	0	1	1	0	1
2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

Impact of Bin Count (M)

- **Power of 2**

13 = (1 * 2^3) + (1 * 2^2) + (1 * 2^0)

31	30	29	28	27	26		5	4	3	2	1	0
0	0	0	0	0	0	…	0	0	1	1	0	1
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}		2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

13 / 2^2 = (1 * 2^1) + (1 * 2^0) + ~~(1 * 2^-2)~~ **>> 2**

31	30	29	28	27	26		5	4	3	2	1	0
0	0	0	0	0	0	…	0	0	0	0	1	1
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}		2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

Impact of Bin Count (M)

- **Power of 2**

$$13 / 2^2 = (1 * 2^1) + (1 * 2^0) + \cancel{(1 * 2^{-2})} = 3$$

31	30	29	28	27	26		5	4	3	2	1	0
0	0	0	0	0	0	…	0	0	0	0	1	1
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}		2^5	2^4	2^3	2^2	2^1	2^0

$$(13 / 2^2) * 2^2 = (1 * 2^3) + (1 * 2^2)$$

<< 2

31	30	29	28	27	26		5	4	3	2	1	0
0	0	0	0	0	0	…	0	0	1	1	0	0
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}		2^5	2^4	2^3	2^2	2^1	2^0

Impact of Bin Count (M)

- **Power of 2**

13 % 2^2 = 13 - (13 / 2^2) * 2^2   c.f) a % n = a − ($\lfloor \frac{a}{n} \rfloor$ × n)

31	30	29	28	27	26		5	4	3	2	1	0
0	0	0	0	0	0	…	0	0	1	1	0	1
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}		2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

13

subtract

31	30	29	28	27	26		5	4	3	2	1	0
0	0	0	0	0	0	…	0	0	1	1	0	0
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}		2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}

(13 / 2^2) * 2^2
= (13 >> 2) << 2

Impact of Bin Count (M)

- **Power of 2**

$13 \% 2\text{^}2 = 13 - (13 / 2\text{^}2) * 2\text{^}2$ c.f) $a \% n = a - (\lfloor \frac{a}{n} \rfloor \times n)$

31	30	29	28	27	26		5	4	3	2	1	0	
0	0	0	0	0	0	…	0	0	1	1	0	1	**13**
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}		2^5	2^4	2^3	2^2	2^1	2^0	

These bits become zero

subtract

31	30	29	28	27	26		5	4	3	2	1	0
0	0	0	0	0	0	…	0	0	1	1	0	0
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}		2^5	2^4	2^3	2^2	2^1	2^0

(13 / 2^2) * 2^2
= (13 >> 2) << 2

Impact of Bin Count (M)

- **Power of 2**

13 % 2^2 = 13 - (13 / 2^2) * 2^2 = 13 – (13 >> 2) << 2 = 13 & (2^2 – 1)

31	30	29	28	27	26		5	4	3	2	1	0	
0	0	0	0	0	0	…	0	0	1	1	0	1	**13**
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}		2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	

bitwise AND

31	30	29	28	27	26		5	4	3	2	1	0	
0	0	0	0	0	0	…	0	0	0	0	1	1	**2^2 - 1**
2^{31}	2^{30}	2^{29}	2^{28}	2^{27}	2^{26}		2^{5}	2^{4}	2^{3}	2^{2}	2^{1}	2^{0}	

Impact of Bin Count (M)

- Power of 2
  - Similar to multiples of 2, bins can become biased
  - But the modulo operation (%) can be replaced by the bitwise AND operation (&)

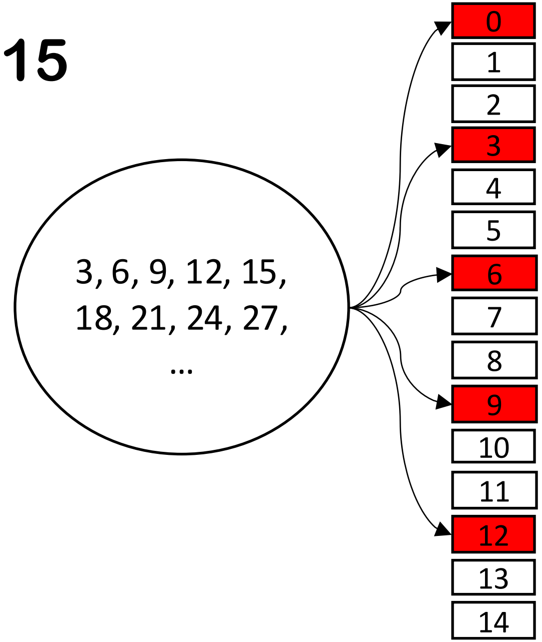
if) M is power of 2

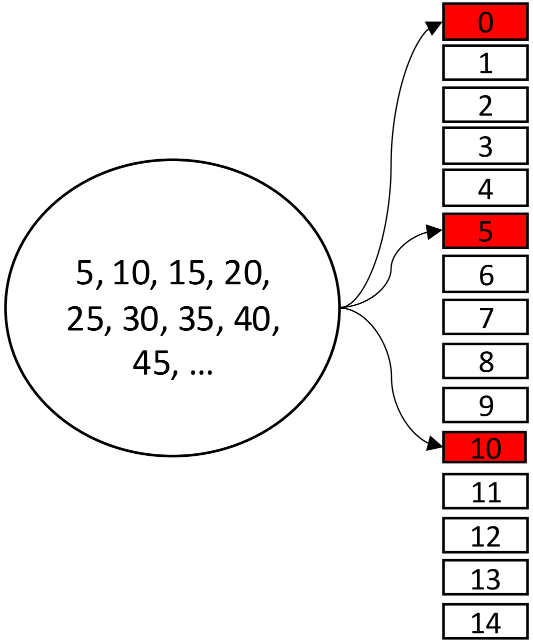
=> K % M = K & (M – 1)

Impact of Bin Count (M)

- **Composite Number**

ex) M is 15

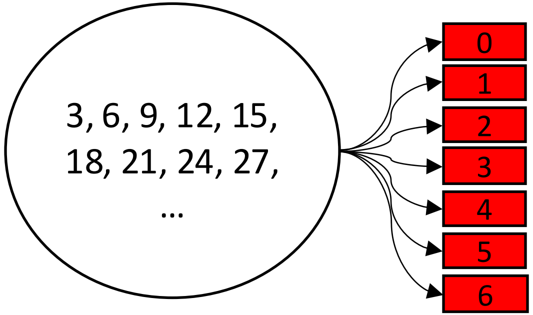


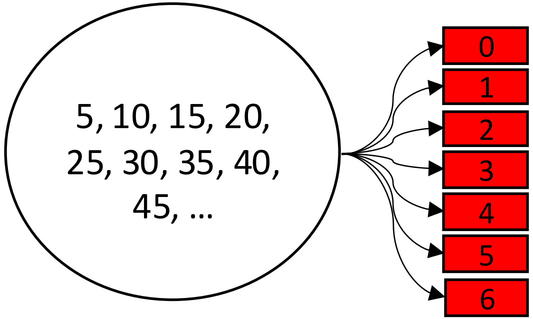


Impact of Bin Count (M)

- **Prime Number**

ex) M is 7





Basic Hash Functions

- Division Hashing

- **Multiplicative Hashing**

Multiplicative Hashing

$\{x\} = x - \lfloor x \rfloor$

Circumference is 1

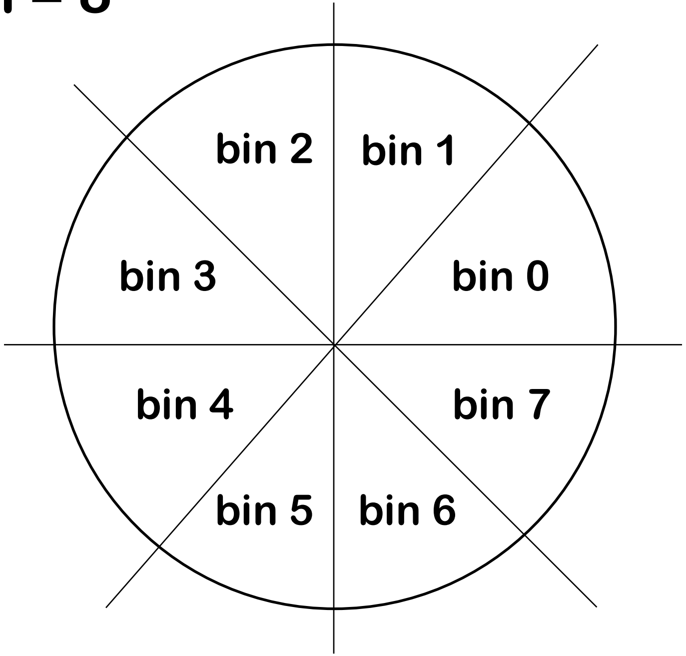
Divide the circle into M segments and mark the following points.

=> $\{1C\}, \{2C\}, \{3C\}, \{4C\}, \ldots$

=> $\{\mathrm{k}C\}$: k is key and C is constant

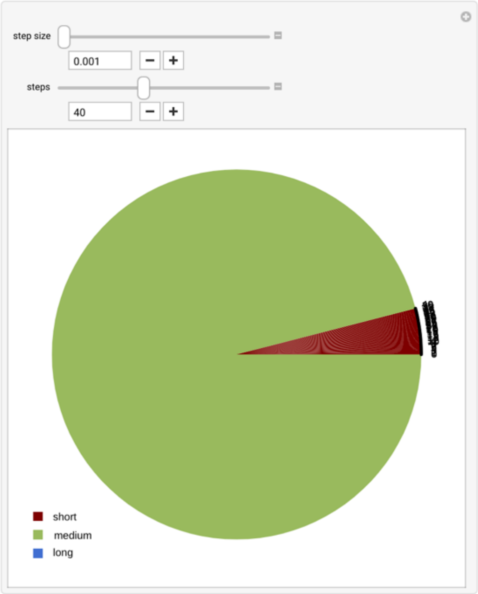
=> <u>Select the bin corresponding to the segment</u>

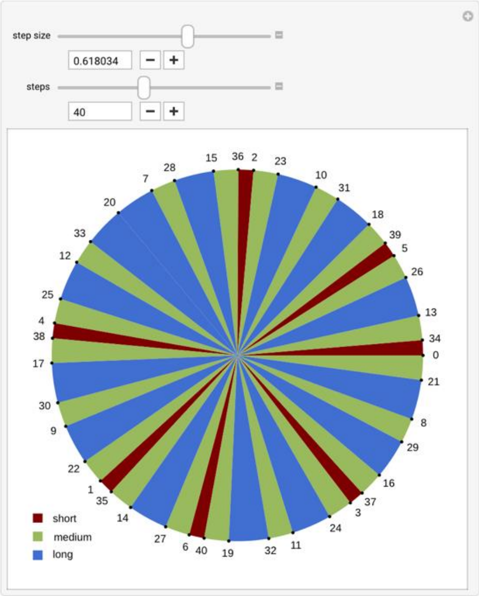
ex) M = 8



Multiplicative Hashing

- https://demonstrations.wolfram.com/ThreeDistanceTheorem/





ex) {C} is Fibonacci Ratio

Multiplicative Hashing

- We want to calculate $\lfloor((key * C) \% 1) * M\rfloor$

- With the minimum cost

Multiplicative Hashing

- Let's define a decimal number as an integer variable
  - Greater error compared to floating-point
  - Faster computation

31	30	29	28	27	26		5	4	3	2	1	0
1	0	0	1	1	1	…	1	1	1	0	0	1
2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}		2^{-27}	2^{-28}	2^{-29}	2^{-30}	2^{-31}	2^{-32}

ex) 0.6180339887

= (1 * 2^-1) + (1 * 2^-4) + (1 * 2^-5) + … + (1 * 2^-28) + (1 * 2^-29) + (1 * 2^-32)

Multiplicative Hashing

- **Imagine there's an integer part**
  - **But in reality, only the fractional part is stored as an int (32bit)**

5	4	3	2	1	0
0	0	0	0	1	1
2^5	2^4	2^3	2^2	2^1	2^0

31	30	29	28	27	26		5	4	3	2	1	0
1	0	0	1	1	1	…	1	1	1	0	0	1
2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}		2^{-27}	2^{-28}	2^{-29}	2^{-30}	2^{-31}	2^{-32}

ex) 3.6180339887

Multiplicative Hashing

- **Let's multiply the constant by a key**
  - **The integer part of the result automatically disappears**

=> ((key * C) % 1)

5	4	3	2	1	0
0	0	0	0	0	0
2^5	2^4	2^3	2^2	2^1	2^0

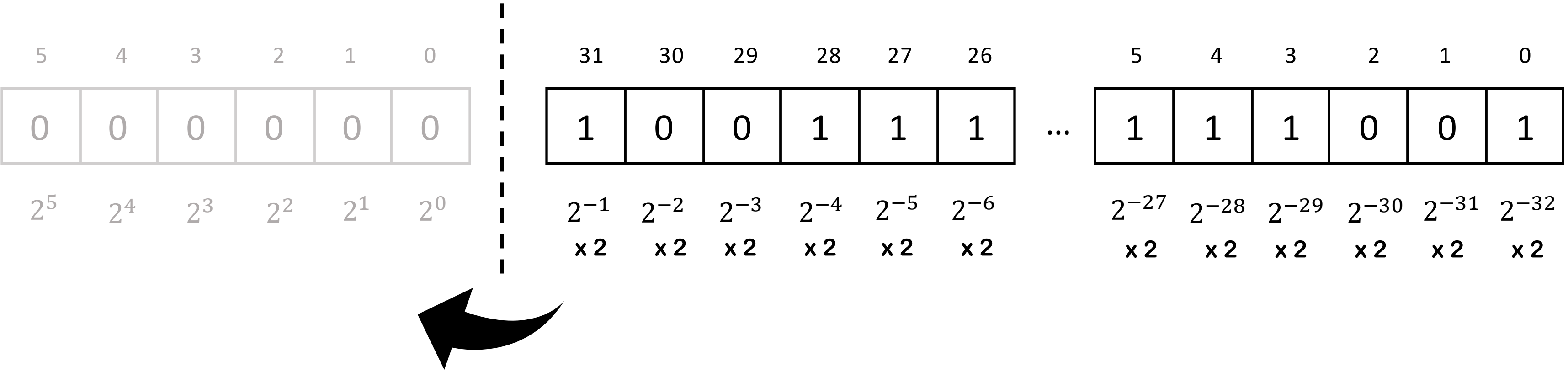
31	30	29	28	27	26		5	4	3	2	1	0
1	0	0	1	1	1	…	1	1	1	0	0	1
2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}		2^{-27}	2^{-28}	2^{-29}	2^{-30}	2^{-31}	2^{-32}

Multiplicative Hashing

- **Let's multiply the constant by a key**
  - **The integer part of the result automatically disappears**

=> ((key * C) % 1)

ex) K = 2



Multiplicative Hashing

- Let's multiply the constant by a key
  - The integer part of the result automatically disappears

=> ((key * C) % 1)

ex) K = 5 = 2^2 + 2^0

5	4	3	2	1	0
0	0	0	0	0	0
2^5	2^4	2^3	2^2	2^1	2^0

31	30	29	28	27	26		5	4	3	2	1	0
1	0	0	1	1	1	…	1	1	1	0	0	1
2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}		2^{-27}	2^{-28}	2^{-29}	2^{-30}	2^{-31}	2^{-32}
x K	x K	x K	x K	x K	x K		x K	x K	x K	x K	x K	x K

Multiplicative Hashing

- **Let's multiply the M**
  - **The integer part of the result automatically disappears (?)**

$$\Rightarrow \lfloor ((key * C) \% 1) * M \rfloor$$

ex) M = 9

5	4	3	2	1	0
0	0	0	0	0	0
2^5	2^4	2^3	2^2	2^1	2^0

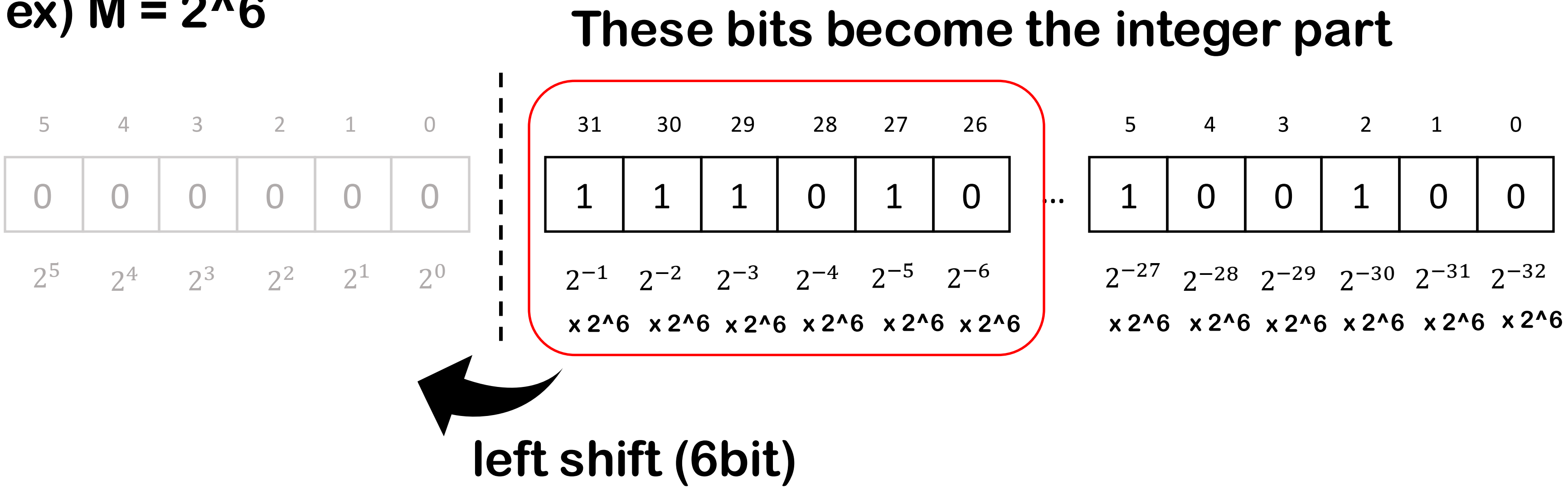
?

31	30	29	28	27	26		5	4	3	2	1	0
1	1	1	0	1	0	…	1	0	0	1	0	0
2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}		2^{-27}	2^{-28}	2^{-29}	2^{-30}	2^{-31}	2^{-32}
x 9	x 9	x 9	x 9	x 9	x 9		x 9	x 9	x 9	x 9	x 9	x 9

Multiplicative Hashing

- Let's set M as a power of 2 (M = 2^m)

ex) M = 2^6



Multiplicative Hashing

- Let's set M as a power of 2 (M = 2^m)
  - Then we can calculate the formula as (Key * C) >> (32 – m)

ex) M = 2^6

These bits become the integer part

5	4	3	2	1	0
0	0	0	0	0	0
2^5	2^4	2^3	2^2	2^1	2^0

31	30	29	28	27	26		5	4	3	2	1	0
1	1	1	0	1	0	…	1	0	0	1	0	0
2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}		2^{-27}	2^{-28}	2^{-29}	2^{-30}	2^{-31}	2^{-32}
x 2^6	x 2^6	x 2^6	x 2^6	x 2^6	x 2^6		x 2^6	x 2^6	x 2^6	x 2^6	x 2^6	x 2^6

>> (32 – 6)

~~left shift~~ (6bit)

Hash Functions

- MurmurHash
- CityHash
- CRC32
- etc…

Thank You