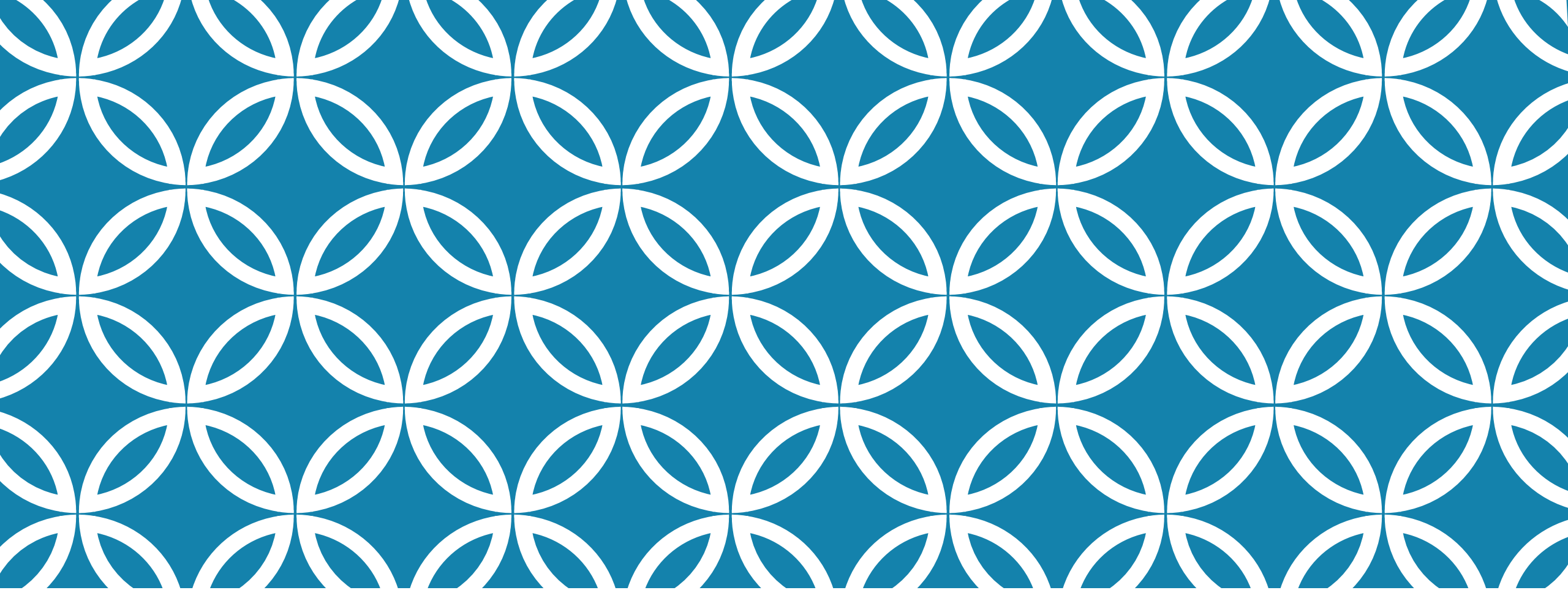
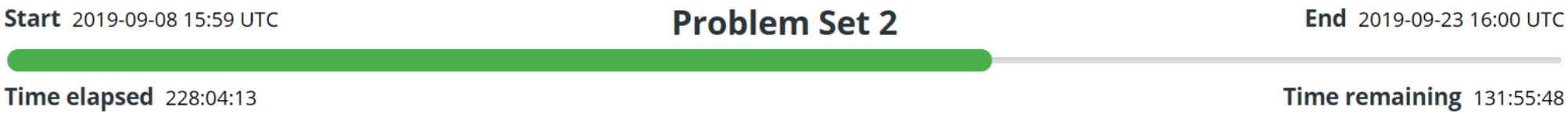


LECTURE 10: HASHING (PART 2)

Harold Soh
harold@comp.nus.edu.sg

ADMINISTRATIVE ISSUES: PROBLEM SET 2



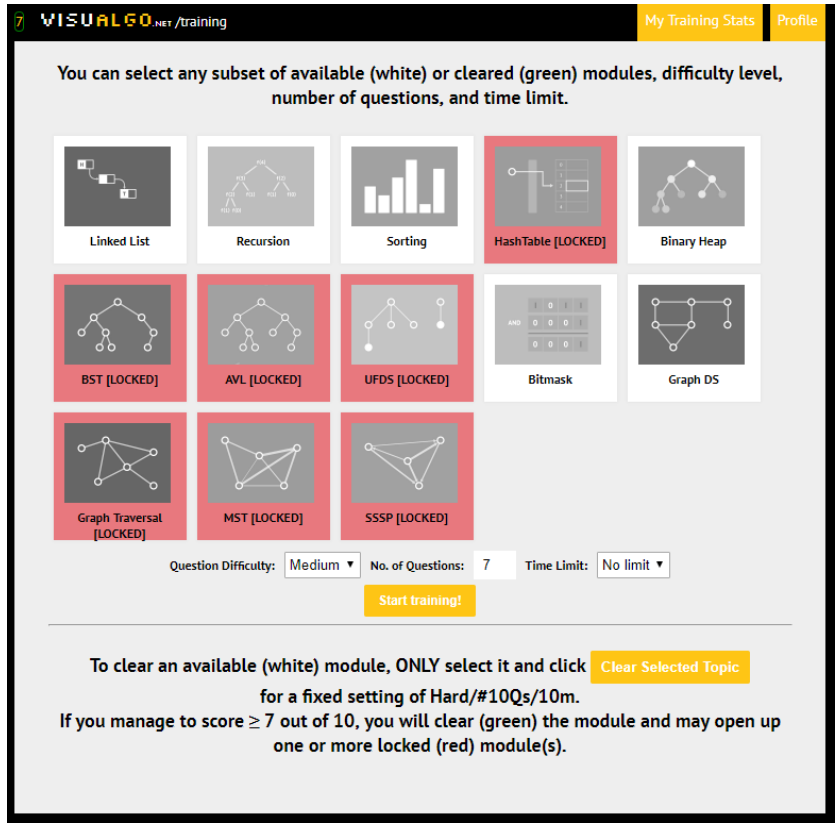
- Deadline is Monday 23rd 23:59:19
- 3 graded problems
  - Kattis Quest
  - Cookie Selection
  - GCPC
- 2 challenge (ungraded) problems
  - BST
  - Shortsell

QUIZ 2

Tuesday after recess (1st Oct)

Covers Material up to **AVL Trees.**

- **NO HASHING/SYMBOL TABLES!**



QUESTIONS?

QUICK RECAP

Dictionary ADT

Hash Tables

Separate Chaining

Open Addressing

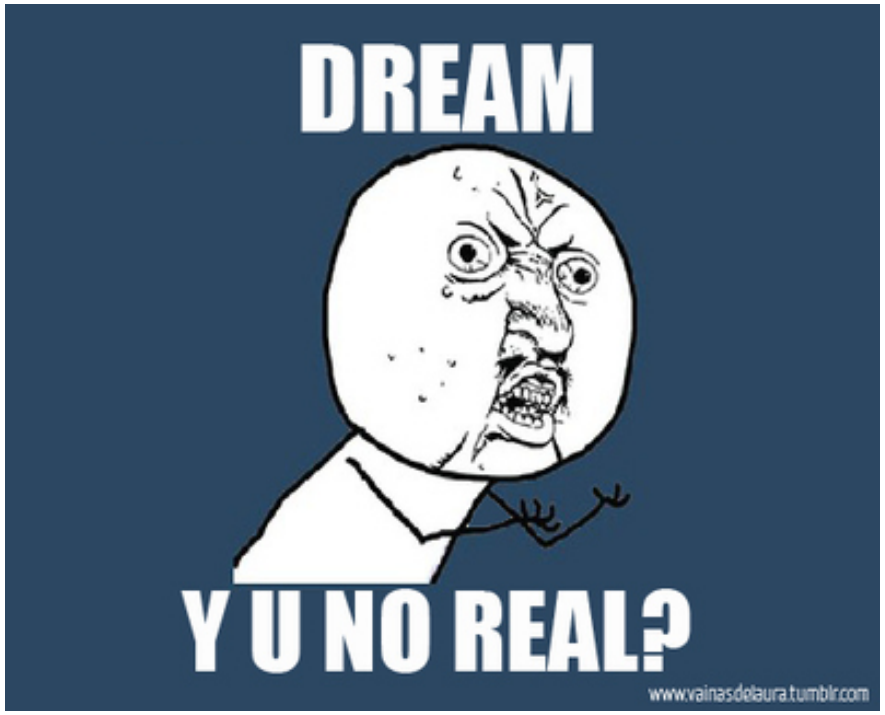
- linear probing
- quadratic probing
- double hashing

SUHA IS A DREAM!

Simple Uniform Hashing doesn't exist (in general).

BUT: Tells us properties of a "good" hashing function:

A. Consistent: same key maps to same bucket.
B. Fast to compute, $O(1)$
C. Scatter the keys into different buckets as uniformly as possible $\in [0..m-1]$



NEXT:

Designing Real Hashing Functions.

Dynamically growing table sizes.

Extra: Cuckoo Hashing (intro)

LEARNING OUTCOMES

By the end of this session, students should be able to:

- Describe the **division and multiplication methods** for generating hashes.
- Explain how to **resize (grow) hash tables dynamically.**

EXAMPLE HASH FUNCTIONS

Let h_{FL}(string) = first letter

- h_{FL}("Naruto") = 'N'
- h_{FL}("monkey") = 'm'
- h_{FL}("supercalifragilisticexpialidocious") = 's'

is h_{FL} a good hash function?

A. Of course yes!
B. No
C. This seems like a simple problem. But is there a trick?

EXAMPLE HASH FUNCTIONS

Let h_{FL}(string) = first letter

- h_{FL}("Naruto") = 'N'
- h_{FL}("monkey") = 'm'
- h_{FL}("supercalifragilisticexpialidocious") = 's'

is the first letter fast to compute? Yes!

is the first letter uniformly distributed?

Heck, No!

is h_{FL} a good hash function?

A. Of course yes!

B. No

Want:

A. Consistent

B. Fast to compute, $O(1)$

C. as uniform as possible $\in [0..m-1]$

EXAMPLE HASH FUNCTIONS

Let h_{FL}(string) = first letter

- h_{FL}("Naruto") = 'N'
- h_{FL}("monkey") = 'm'
- h_{FL}("supercalifragilisticexpialidocious") = 's'

is the first letter fast to compute? Yes!

is the first letter uniformly distributed?

Heck, No!

Letter	Relative frequency as the first letter of an English word
t	15.978%
a	11.682%
o	7.631%
i	7.294%
s	6.686%
w	5.497%
c	5.238%
b	4.434%
p	4.319%
h	4.200%
f	4.027%
m	3.826%
d	3.174%
r	2.826%
e	2.799%
l	2.415%
n	2.284%
g	1.642%
u	1.183%
v	0.824%
y	0.763%
j	0.511%
k	0.456%
q	0.222%
x	0.045%
z	0.045%

MAYBE WE TRY THIS?

Let h_{SUM}(string) = sum of the letters

- h_{SUM}("hat") = 8 + 1 + 20 = 29
- h_{SUM}("cat") = 3 + 1 + 20 = 24

Want:

A. Consistent
B. Fast to compute, $O(1)$
C. as uniform as possible $\in [0..m-1]$

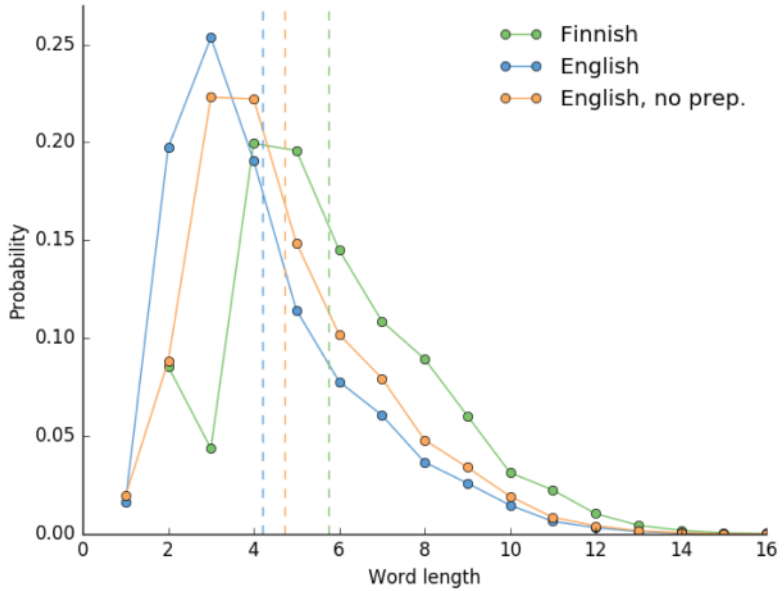
is h_{SUM} a good hash function?

A. Of course yes!
B. No
C. Umm.. maybe.. It sums up a whole bunch of stuff.

MAYBE WE TRY THIS?

Let h_{SUM}(string) = sum of the letters

- h_{SUM}("hat") = 8 + 1 + 20 = 29
- h_{SUM}("cat") = 3 + 1 + 20 = 24



is h_{SUM} a good hash function?

A. Of course yes!

B. No

C. Umm.. maybe.. It sums up a whole bunch of stuff.

Also, all anagrams have the same hash e.g., god, dog

DESIGNING HASH FUNCTIONS

Want: Hash function whose values *look* random

Similar to pseudorandom number generators

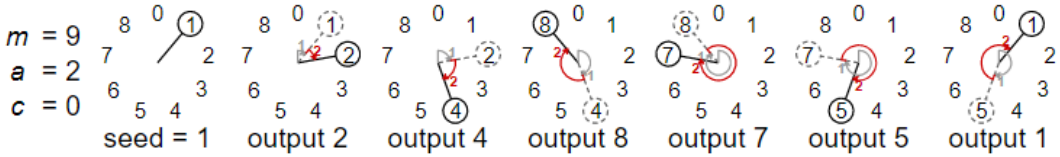
Two common hashing techniques:

- Division Method
- Multiplication Method

A **linear congruential generator** (**LCG**) pseudorandom number generator:

$$x_{n+1} = (ax_n + c) \bmod m$$

For special choices of a, c and m, LCGs can produce numbers that pass formal tests of randomness.



DIVISION METHOD

$h(k) = k \bmod m$

- $m = 7, \quad h(17) = 3$
- $m = 20, \; h(100) =$ ____
- $m = 20, \; h(97) =$ ____
- $m = 13, \quad h(102) =$ ____

Two keys k_1 and k_2 **collide** when:
$k_1 = k_2 \pmod m$

Want:

A. Consistent
B. Fast to compute, $O(1)$
C. as uniform as possible $\in [0..m-1]$

DIVISION METHOD

$h(k) = k \bmod m$

- $m = 7, \quad h(17) = 3$
- $m = 20, \; h(100) = 0$
- $m = 20, \; h(97) = \_\_\_\_$
- $m = 13, \quad h(102) = \_\_\_\_$

Two keys k_1 and k_2 **collide** when:
$k_1 = k_2 \pmod m$

Want:

A. Consistent
B. Fast to compute, $O(1)$
C. as uniform as possible $\in [0..m-1]$

DIVISION METHOD

$h(k) = k \bmod m$

- $m = 7, \quad h(17) = 3$
- $m = 20, \; h(100) = 0$
- $m = 20, \; h(97) = 17$
- $m = 13, \quad h(102) = \_\_\_\_$

Two keys k_1 and k_2 **collide** when:
$k_1 = k_2 \pmod m$

Want:

A. Consistent
B. Fast to compute, $O(1)$
C. as uniform as possible $\in [0..m-1]$

DIVISION METHOD

$h(k) = k \bmod m$

- $m = 7, \quad h(17) = 3$
- $m = 20, \; h(100) = 0$
- $m = 20, \; h(97) = 17$
- $m = 13, \quad h(102) = 12$

Two keys k_1 and k_2 **collide** when:
$k_1 = k_2 \pmod m$

Want:

A. Consistent
B. Fast to compute, $O(1)$
C. as uniform as possible $\in [0..m-1]$

How to choose m?

CHOICE OF $m = 2^x$

Consider $m = 2^x$

Nice property: very fast to calculate $k \bmod m$.

Why?

Want:

A. Consistent
B. Fast to compute, $O(1)$
C. as uniform as possible $\in [0..m-1]$

CHOICE OF $m = 2^x$

Poll Everywhere

https://bit.ly/2LvG9bq

Consider $m = 2^x$

Nice property: very fast to calculate $k \bmod m$.

Can calculate division and multiplication by 2 via shifts:

- 9/2: 001001 >> 1 = 00100 (4)
- 9/4: 001001 >> 2 = 0010 (2)
- 9x2: 001001 << 1 = 0010010 (18)
- 9x4: 001001 << 2 = 00100100 (36)

How can I use shifts to compute k mod m? Which of the following formulas would work?

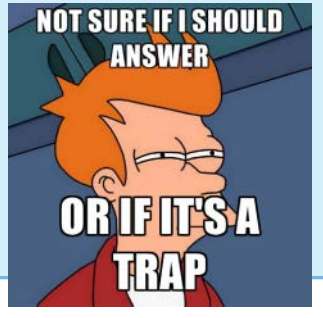
A. $k << x$

B. $k << x - m$

C. $k - ((k << x) >> x)$

D. $k - ((k >> x) << x)$

E.



CHOICE OF $m = 2^x$

Poll Everywhere

https://bit.ly/2LvG9bq

Consider $m = 2^x$

Nice property: very fast to calculate $k \bmod m$.

Can calculate division and multiplication by 2 via shifts:

- 9/2: 001001 >> 1 = 00100 (4)
- 9/4: 001001 >> 2 = 0010 (2)
- 9x2: 001001 << 1 = 0010010 (18)
- 9x4: 001001 << 2 = 00100100 (36)

Fast to compute!

How can I use shifts to compute k mod m? Which of the following formulas would work?

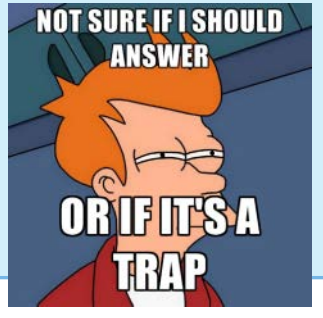
A. $k << x$

B. $k << x - m$

C. $k - ((k << x) >> x)$

D. $\boldsymbol{k - ((k >> x) << x)}$

E.



CHOICE OF $m = 2^x$

Want:

A. Consistent
B. Fast to compute, $O(1)$
C. as uniform as possible $\in [0..m-1]$

Does it satisfy the desirability C?

Does using $m = 2^k$ scatter the keys into different buckets as uniformly as possible?

A. Yes
B. No
C. Yes... No... Maybe... I don't know.
D. Ask Naruto:

CHOICE OF $m = 2^x$

Want:

A. Consistent
B. Fast to compute, $O(1)$
C. as uniform as possible $\in [0..m-1]$

Does it satisfy the desirability C?

Why?

Does using $m = 2^k$ scatter the keys into different buckets as uniformly as possible?

A. Yes
B. No
C. Yes... No... Maybe... I don't know.
D. Ask Naruto:

CHOICE OF $m = 2^x$

Want:

A. Consistent
B. Fast to compute, $O(1)$
C. as uniform as possible $\in [0..m-1]$

Does it satisfy the desirability C?

Problem: Regularity of keys

- Input keys are *often* not uniformly distributed

If all input keys are even:

- $h(k) = k \bmod m$ must be even.

Why?

CHOICE OF $m = 2^x$

Want:

A. Consistent
B. Fast to compute, $O(1)$
C. as uniform as possible $\in [0..m-1]$

Does it satisfy the desirability C?

Problem: Regularity of keys

- Input keys are *often* not uniformly distributed

If all input keys are even:

- $h(k) = k \bmod m$ must be even.

even even ?

$$k = \underbrace{im}_{\text{some multiple of } m} + \underbrace{k \bmod m}_{\text{remainder}}$$

CHOICE OF $m = 2^x$

Want:

A. Consistent
B. Fast to compute, $O(1)$
C. as uniform as possible $\in [0..m-1]$

Does it satisfy the desirability C?

Problem: Regularity of keys

- Input keys are *often* not uniformly distributed

If all input keys are even:

- $h(k) = k \bmod m$ must be even.

even even **even**

$$k = \underbrace{im}_{\text{some multiple of } m} + \underbrace{k \bmod m}_{\text{remainder}}$$

CHOICE OF $m = 2^x$

Want:

A. Consistent
B. Fast to compute, $O(1)$
C. as uniform as possible $\in [0..m-1]$

Does it satisfy the desirability C?

Problem: Regularity of keys

- Input keys are *often* not uniformly distributed

If all input keys are even:

- $h(k) = k \bmod m$ must be even.

No longer uniform: will waste ½ the space and increase collisions!

Fast to compute, but does not use table well.
Also, will only use lowest order bits of k (Why?)

Poll Everywhere

https://bit.ly/2LvG9bq

REGULARITY AND COMMON DIVISORS

Division method: $h(k) = k \bmod m$

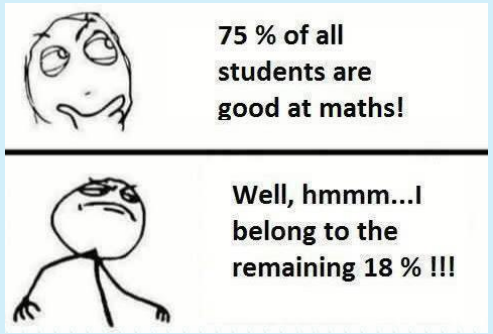
If k and m have a common divisor d

then:

$$\underbrace{k}_{\text{divisible by } d} = \underbrace{im}_{\text{divisible by } d} + \underbrace{k \bmod m}_{?}$$

Assume chaining. How much of the table do we use if both x and m have a common divisor d?

A. d
B. $2\mathrm{d}$
C. $1/d^2$
D. $1/d$
E.



Poll Everywhere

https://bit.ly/2LvG9bq

REGULARITY AND COMMON DIVISORS

Division method: $h(k) = k \bmod m$

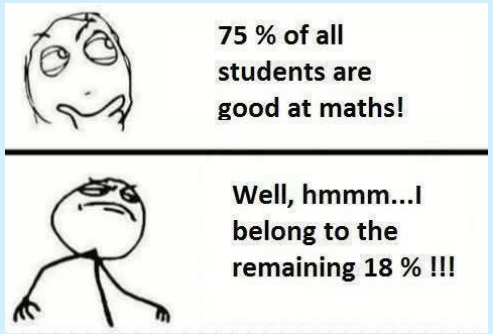
If k and m have a common divisor d

then:

$$\underbrace{k}_{\text{divisible by } d} = \underbrace{im}_{\text{divisible by } d} + \underbrace{k \bmod m}_{\text{divisible by } d}$$

Assume chaining. How much of the table do we use if both x and m have a common divisor d?

A. d
B. 2d
C. $1/d^2$
D. $\boldsymbol{1/d}$
E.



REGULARITY AND COMMON DIVISORS

Division method: $h(k) = k \bmod m$

If k and m have a common divisor d

then:

$$\underbrace{k}_{\text{divisible by } d} = \underbrace{im}_{\text{divisible by } d} + \underbrace{k \bmod m}_{\text{divisible by } d}$$

Choose m such that it has no common factors with any k

will only use 1 out of every d slots!

0	$(k_1,$ A)
1	null
2	null
d=3	$(k_2,$ B)
4	null
5	null
2d=6	$(k_3,$ C)
7	null
8	null
3d=9	$(k_4,$ D)

DIVISION METHOD

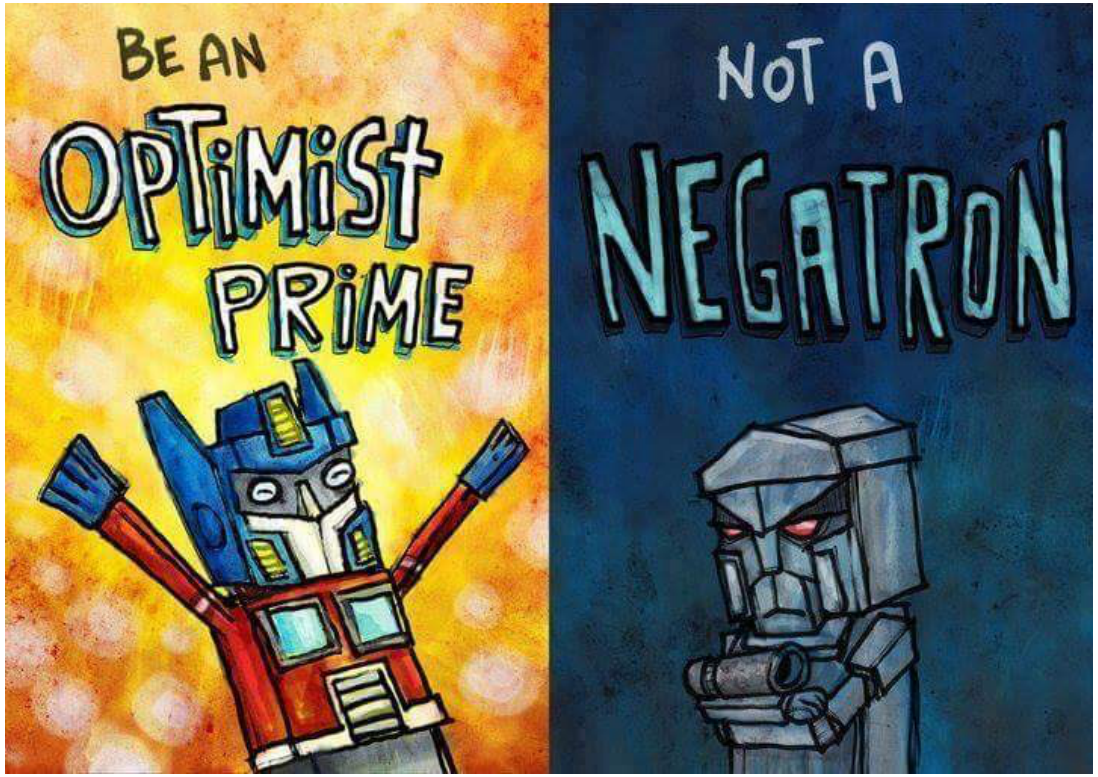
Choose m to be **prime**

- Avoid powers of 2 and powers of 10

In practice: popular and easy

But not always the most effective.

Slow (no more shifts)



DESIGNING HASH FUNCTIONS

Want: Hash function whose values *look* random

Similar to pseudorandom number generators

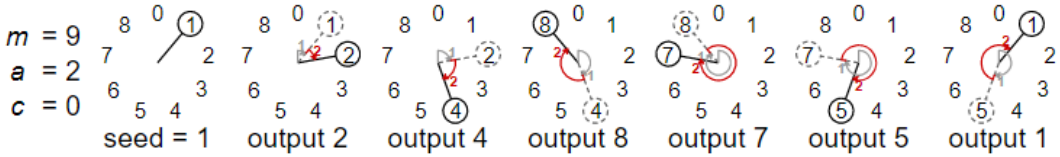
Two common hashing techniques:

- Division Method
- Multiplication Method

A **linear congruential generator** (**LCG**) pseudorandom number generator:

$$x_{n+1} = (ax_n + c) \bmod m$$

For special choices of a, c and m, LCGs can produce numbers that pass formal tests of randomness.



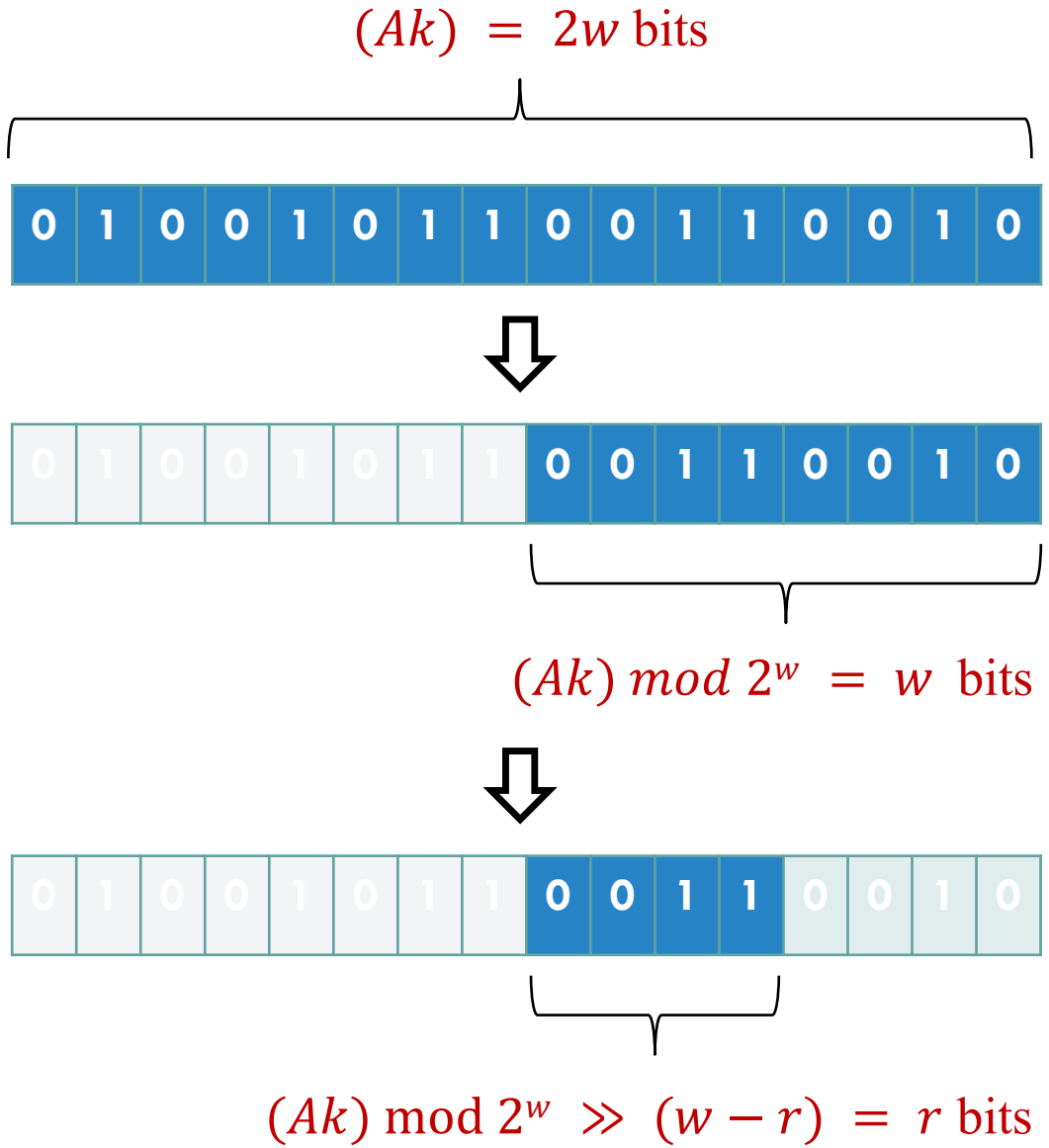
MULTIPLICATION METHOD

Fix

- table size: $m = 2^r$
- word size: w (size of a key in bits)
- constant: $2^{w-1} < A < 2^w$

Then:

$$h(k) = (Ak) \bmod 2^w \gg (w - r)$$



MULTIPLICATION METHOD

Fix

- table size: $m = 2^r$
- word size: w (size of a key in bits)
- constant: $2^{w-1} < A < 2^w$

Then:

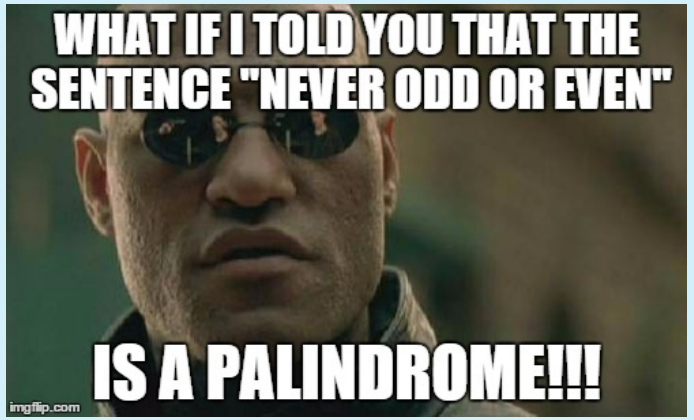
$$h(k) = (Ak) \bmod 2^w \gg (w - r)$$

Poll Everywhere

https://bit.ly/2LvG9bq

For the multiplication method, should A be even or odd?

A. even
B. odd
C. neither
D.



MULTIPLICATION METHOD

Fix

- table size: $m = 2^r$
- word size: w (size of a key in bits)
- constant: $2^{w-1} < A < 2^w$

Then:

$$h(k) = (Ak) \bmod 2^w \gg (w - r)$$

Consider what happens when A is even.

Poll Everywhere

https://bit.ly/2LvG9bq

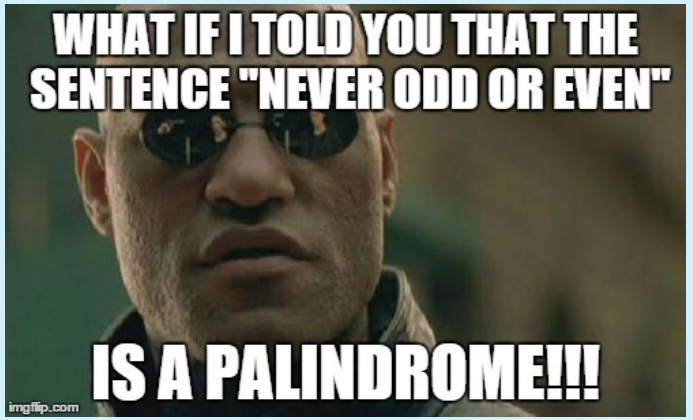
For the multiplication method, should A be even or odd?

A. even

B. odd

C. neither

D.



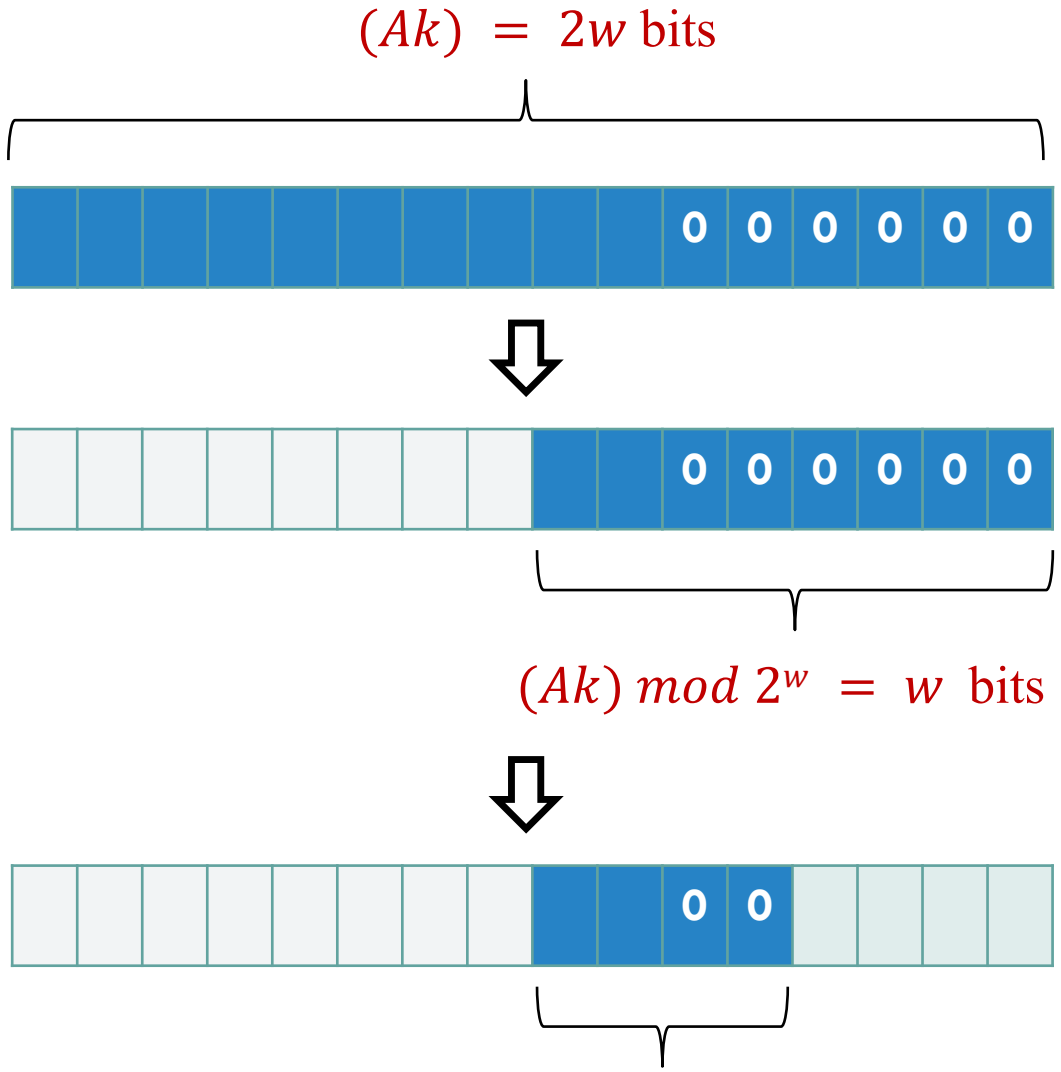
MULTIPLICATION METHOD

Fix

- table size: $m = 2^r$
- word size: w (size of a key in bits)
- constant: $2^{w-1} < A < 2^w$

Then:

$$h(k) = (Ak) \bmod 2^w \gg (w - r)$$



Consider what happens when A is even, say $2^{w-1} + 64$. (see example in the right)

Point: even numbers cause at least one bit of information loss.

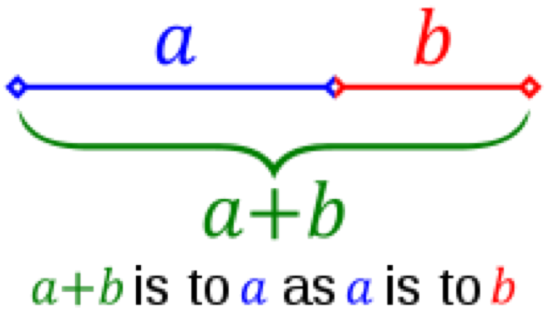
MULTIPLICATION METHOD

Choose A, r carefully

In practice: works well with A is chosen well (e.g., odd)

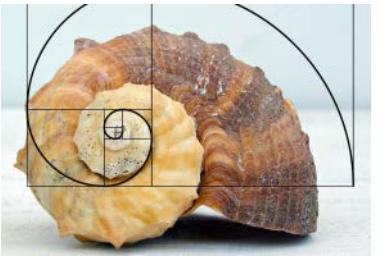
Knuth recommends $A \approx \frac{\sqrt{5}-1}{2} \cdot 2^{32}$ for $w = 32$ bit words

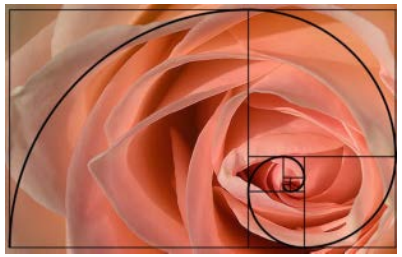
Very Fast (shifts!)



Donald Knuth

wrote The Art of Computer Programming (TAOCP)





DESIGNING HASH FUNCTIONS

Want: Hash function whose values *look* random

Similar to pseudorandom number generators

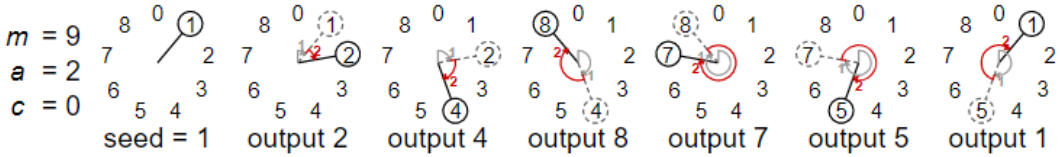
Two common hashing techniques:

- Division Method
- Multiplication Method

A **linear congruential generator** (**LCG**) pseudorandom number generator:

$$x_{n+1} = (ax_n + c) \bmod m$$

For special choices of a, c and m, LCGs can produce numbers that pass formal tests of randomness.



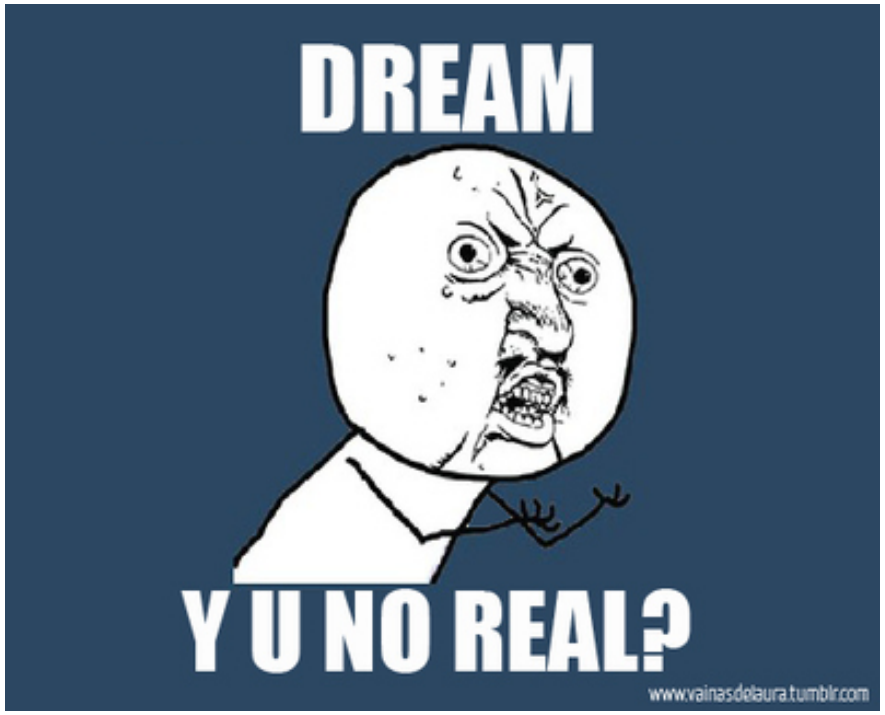
SUHA IS A DREAM!

But we can get pretty close in practice!

Simple Uniform Hashing doesn't exist (in general).

BUT: Tells us properties of a "good" hashing function:

A. Consistent: same key maps to same bucket.
B. Fast to compute, $O(1)$
C. Scatter the keys into different buckets as uniformly as possible $\in [0..m-1]$



SOME TAKE-AWAYS

Choosing a proper hash function is important

In general, good hash functions are **hard to design**.

Have to choose the table size m properly.

HOW TO SET THE TABLE SIZE?

Assume:

- Hashing with Chaining
- Simple Uniform Hashing
- Expected Search Time: $O\left(1+\frac{n}{m}\right)$
- Optimal size: $m = cn$
  - if $m < 2n$: ???

HOW TO SET THE TABLE SIZE?

Assume:

- Hashing with Chaining
- Simple Uniform Hashing
- Expected Search Time: $O\left(1+\frac{n}{m}\right)$
- Optimal size: $m = cn$
  - if $m < 2n$: too many collisions

HOW TO SET THE TABLE SIZE?

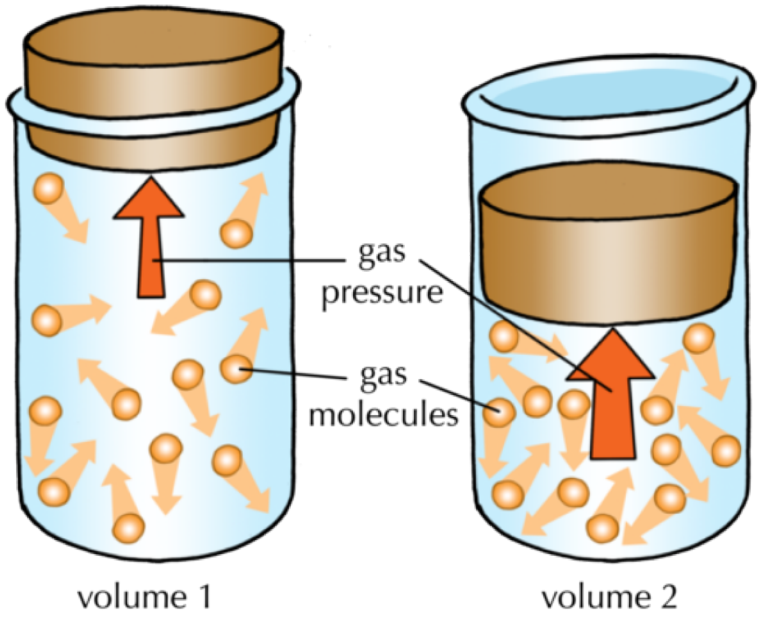
Assume:

- Hashing with Chaining
- Simple Uniform Hashing
- Expected Search Time: $O\left(1+\frac{n}{m}\right)$
- Optimal size: $m = cn$
  - if $m < 2n$: too many collisions
  - if $m > 10n$: ???

HOW TO SET THE TABLE SIZE?

Assume:

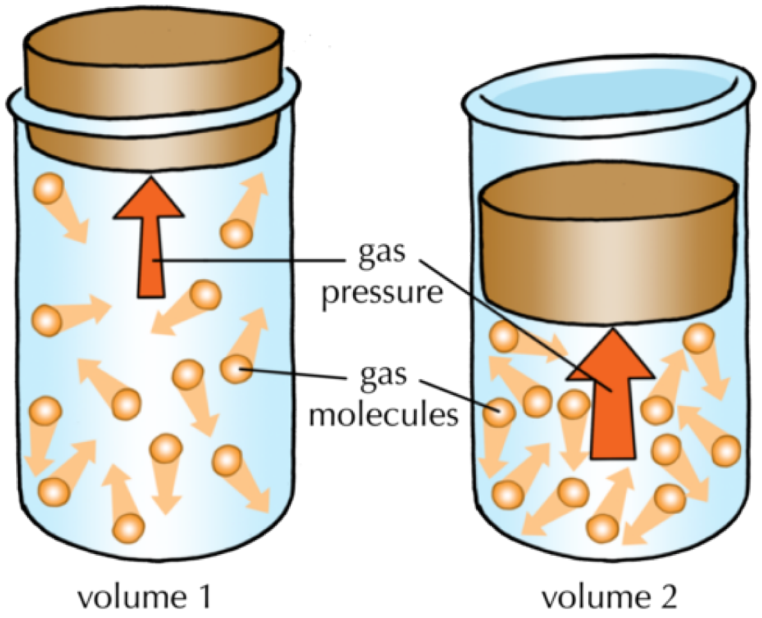
- Hashing with Chaining
- Simple Uniform Hashing
- Expected Search Time: $O\left(1+\frac{n}{m}\right)$
- Optimal size: $m = cn$
  - if $m < 2n$: too many collisions
  - if $m > 10n$: too much wasted space



HOW TO SET THE TABLE SIZE?

Assume:

- Hashing with Chaining
- Simple Uniform Hashing
- Expected Search Time: $O\left(1+\frac{n}{m}\right)$
- Optimal size: $m = cn$
  - if $m < 2n$: too many collisions
  - if $m > 10n$: too much wasted space
- **But:** we don't know n in advance.



DYNAMIC TABLE SIZES!

Idea: Grow & shrink the table size as necessary

Does this idea seem similar?

from Lecture 2

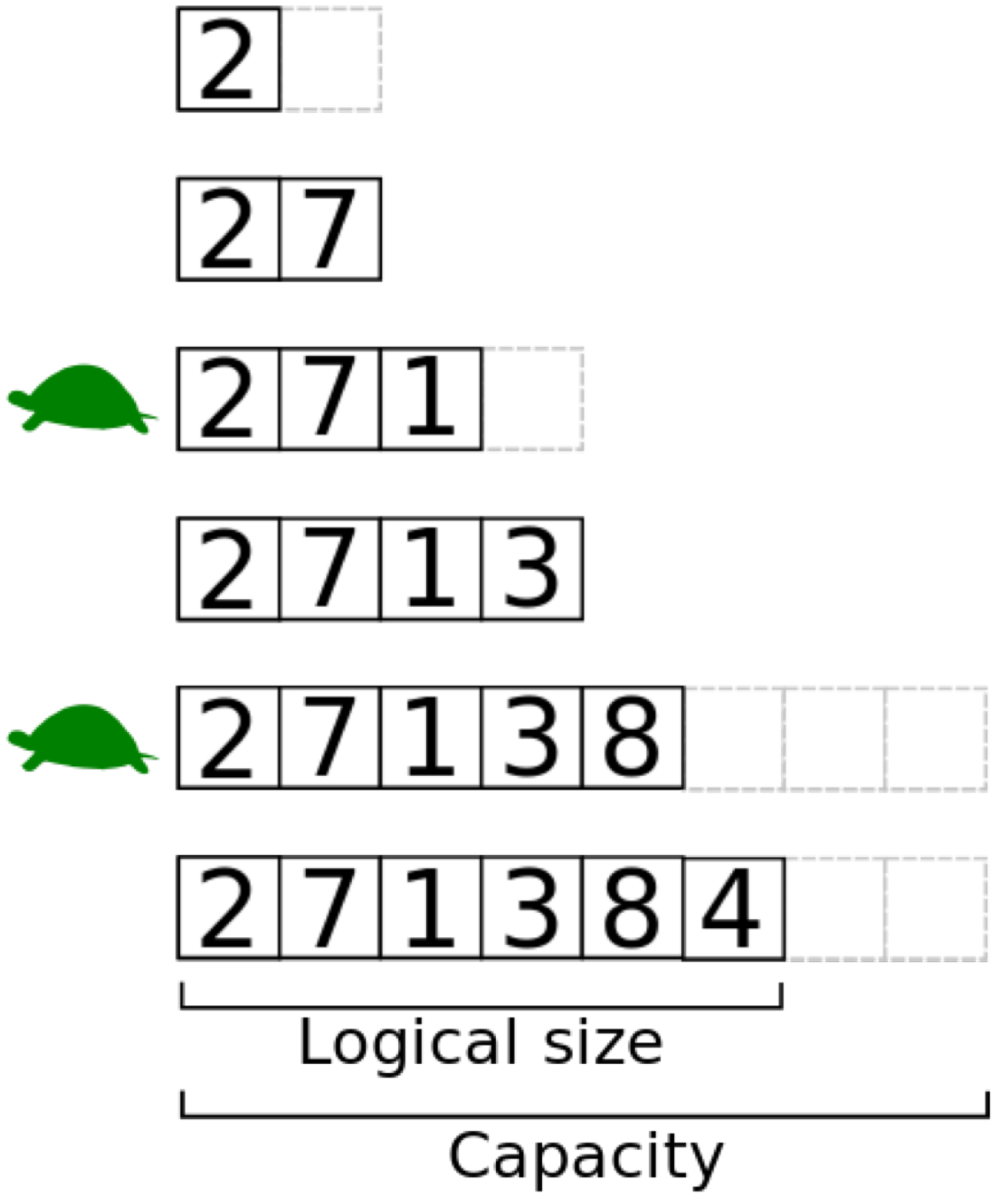
DYNAMIC ARRAYS

Arrays that can grow.

Amortized constant time for adding at the end

Different strategies for growing, e.g.

- Double the space each time you need to grow
- increase by a pre-set amount

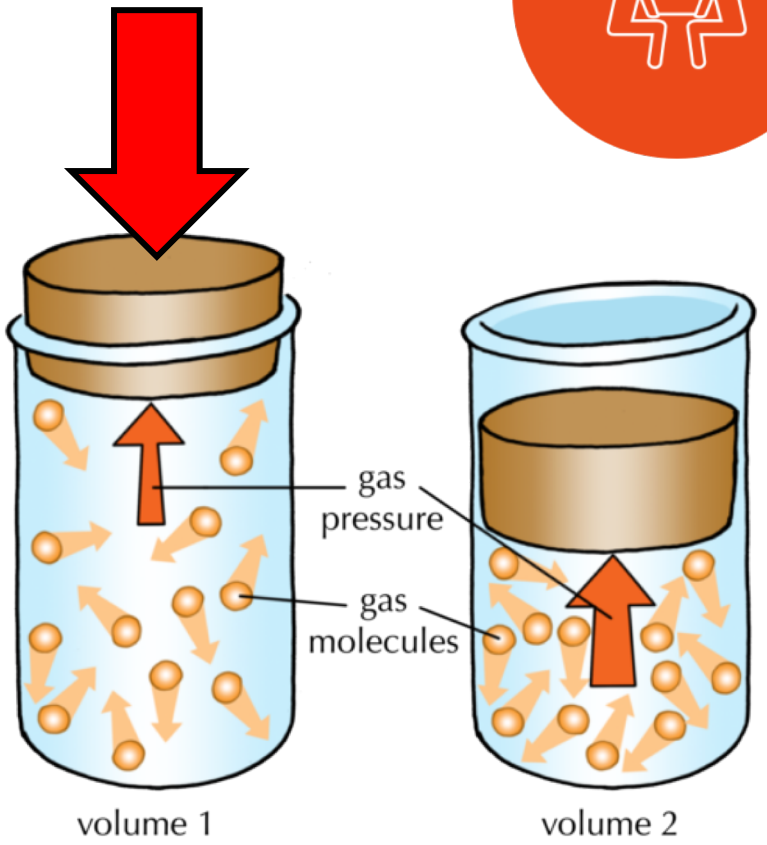


DYNAMIC TABLE SIZES!

Idea: Grow & shrink the table size as necessary

Example:

- Initially, $m_1 = 10$
- After inserting 6 items, table is too small: Grow
- After deleting $n - 1$ items, table too big: Shrink



GROWING THE TABLE

1. Choose new table size $m_2 > m_1$
2. Copy all elements from old table to new table.

This will not work! Why?

hash function depends on table size!

GROWING THE TABLE

1. Choose new table size $m_2 > m_1$
2. Choose a new hash function h_2 that matches m_2
3. For each key-value pair k_i, v_i in the old hash table:
    1. Compute new bucket $b_2 = h_2(k_i)$
    2. Insert (k_i, v_i) into bucket b_2

GROWING THE TABLE

1. Choose new table size $m_2 > m_1$
2. Choose a new hash function h_2 that matches m_2
3. For each key-value pair k_i, v_i in the old hash table:
   1. Compute new bucket $b_2 = h_2(k_i)$
   2. Insert (k_i, v_i) into bucket b_2

Costs:

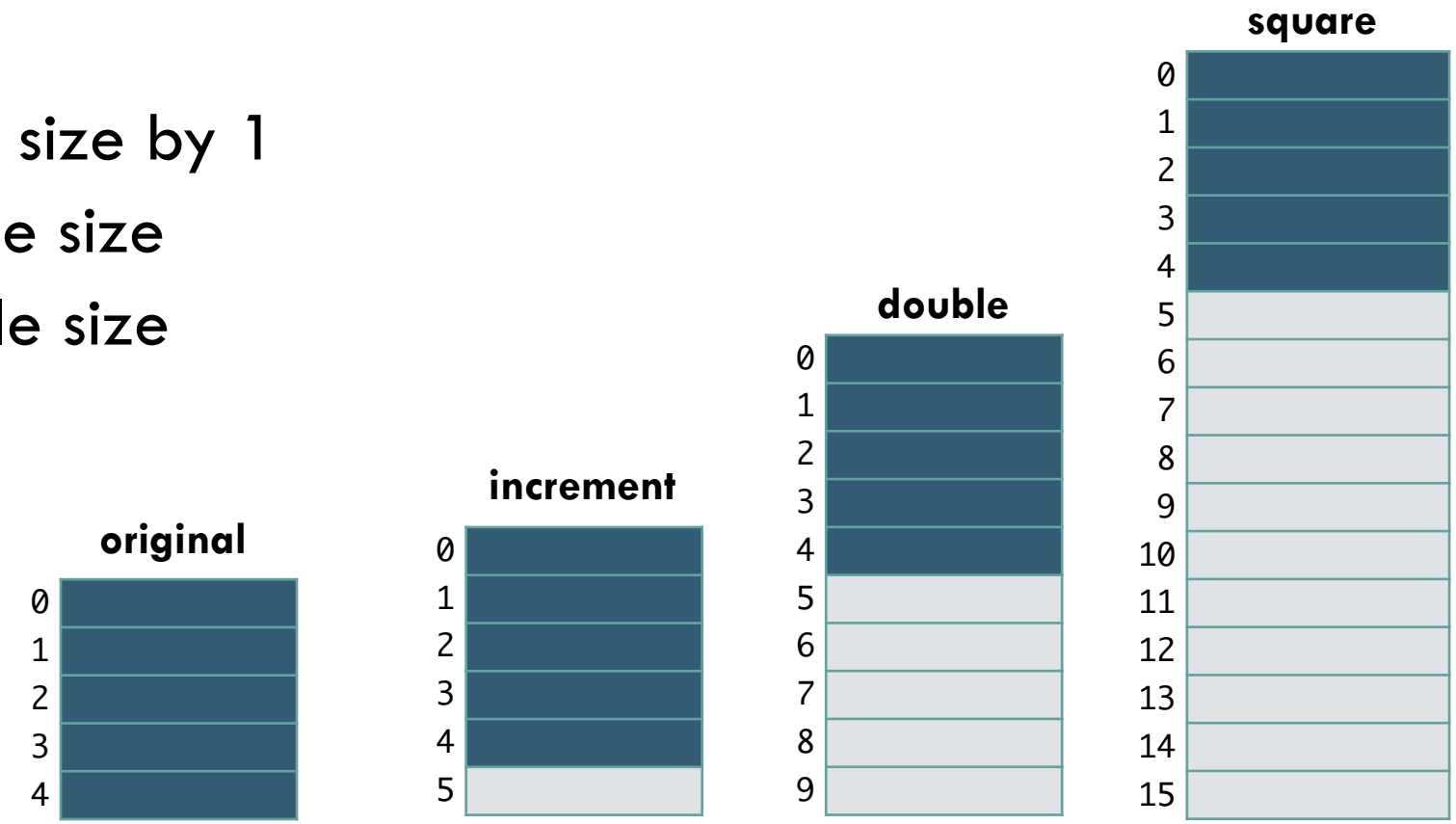
- Creating new table: $O(m_2)$
- Scanning old hash table: $O(m_1)$
- Inserting all n elements into the new hash table: $O(n)$
- Total: $O(m_1 + m_2 + n)$

- **If** $m_2 > m_1$ then $O(m_2 + n)$
- And **if** $m_2 = cn$ then $O(n)$

HOW FAST SHOULD I GROW?

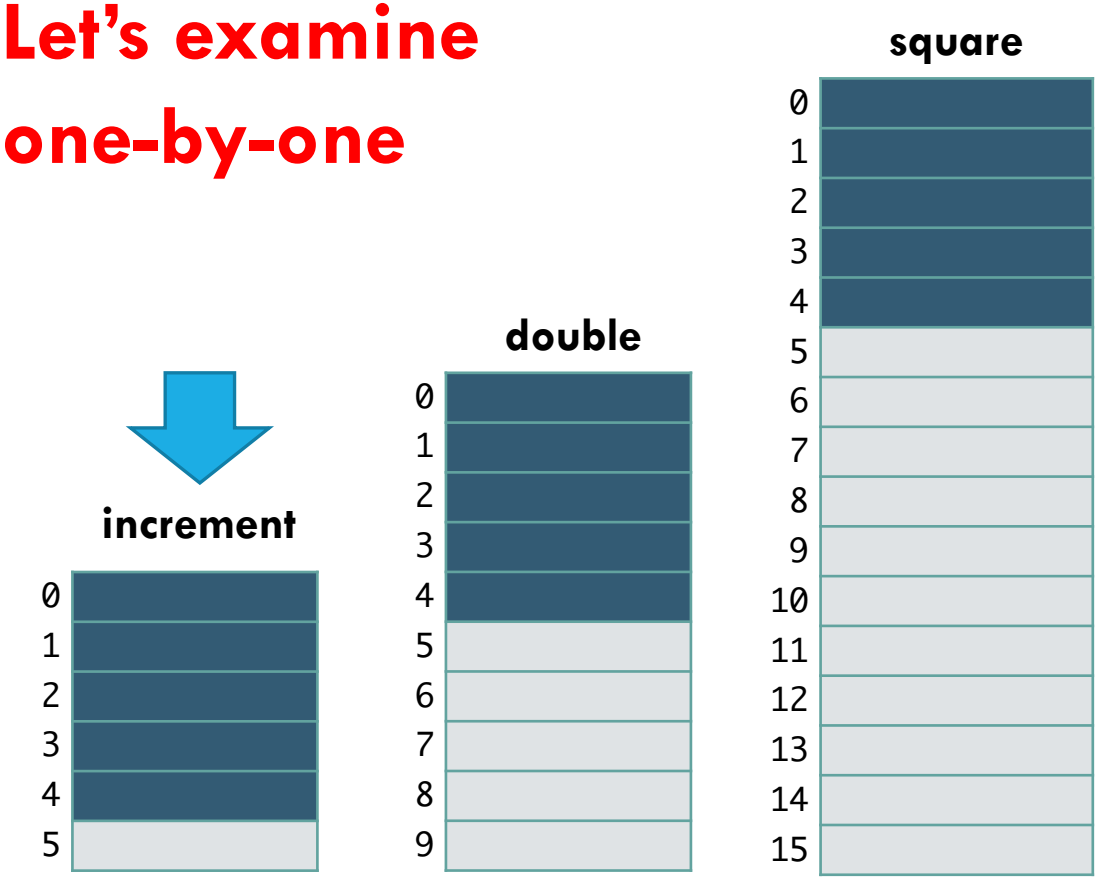
Ideas:

- Increment table size by 1
- Double the table size
- Square the table size



GROWING THE TABLE

Let's examine one-by-one



Which do you think is the better technique?

A. increment by 1
B. double the size
C. square the size
D. They are all the same
E. Definitely not the same, but I don't know which one.

ANALYSIS GUIDE

To keep things simple, we assume:

- We only grow when the table is full, i.e., $n = m$
  - In practice, we should try to keep a low load factor, e.g., grow when 50% full
- We either:
  - Increment by 1
  - Double the size
  - Square the size

INCREMENT BY 1

Hint: You'll probably need:
$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1)$

What is c_i (the cost of the i^{th} insert)?

- If $n < m$ (table is not full), then $c_i = 1$
- If $n = m$ (table is full), then expand so, $c_i = i + (i-1) + 1 = 2i$
- Let's say m_0 is the initial size of the table

What is the Total Cost to insert n elements where $n > m_0$?

$O(n^2)$.

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{m_0-1} c_i + \sum_{i=m_0}^{n} c_i$$
$$\leq m_0 + \sum_{i=m_0}^{n} 2i$$
$$\leq n + 2\sum_{i=m_0}^{n} i$$
$$< n + 2\sum_{i=1}^{n} i$$
$$= n + n(n+1) = O(n^2)$$

INCREMENT BY 1

Hint: You'll probably need:
$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1)$

What is c_i (the cost of the i^{th} insert)?

- If $n < m$ (table is not full), then $c_i = 1$
- If $n = m$ (table is full), then expand so, $c_i = i + (i-1) + 1 = 2i$
- Let's say m_0 is the initial size of the table

What is the <u>average</u> (*amortized*) cost per insert over the sequence?

$O(n)$. Why?

Avg cost: $\frac{1}{n}\sum_{i=1}^{n} c_i \leq \frac{1}{n}\left(n + n(n+1)\right) = n + 2 = O(n)$

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{m_0-1} c_i + \sum_{i=m_0}^{n} c_i$$
$$\leq m_0 + \sum_{i=m_0}^{n} 2i$$
$$\leq m_0 + 2\sum_{i=m_0}^{n} i$$
$$< n + 2\sum_{i=1}^{n} i$$
$$= n + n(n+1) = O(n^2)$$

Hint: You'll probably need:
$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1)$

INCREMENT BY 1

What is c_i (the cost of the i^{th} insert)?

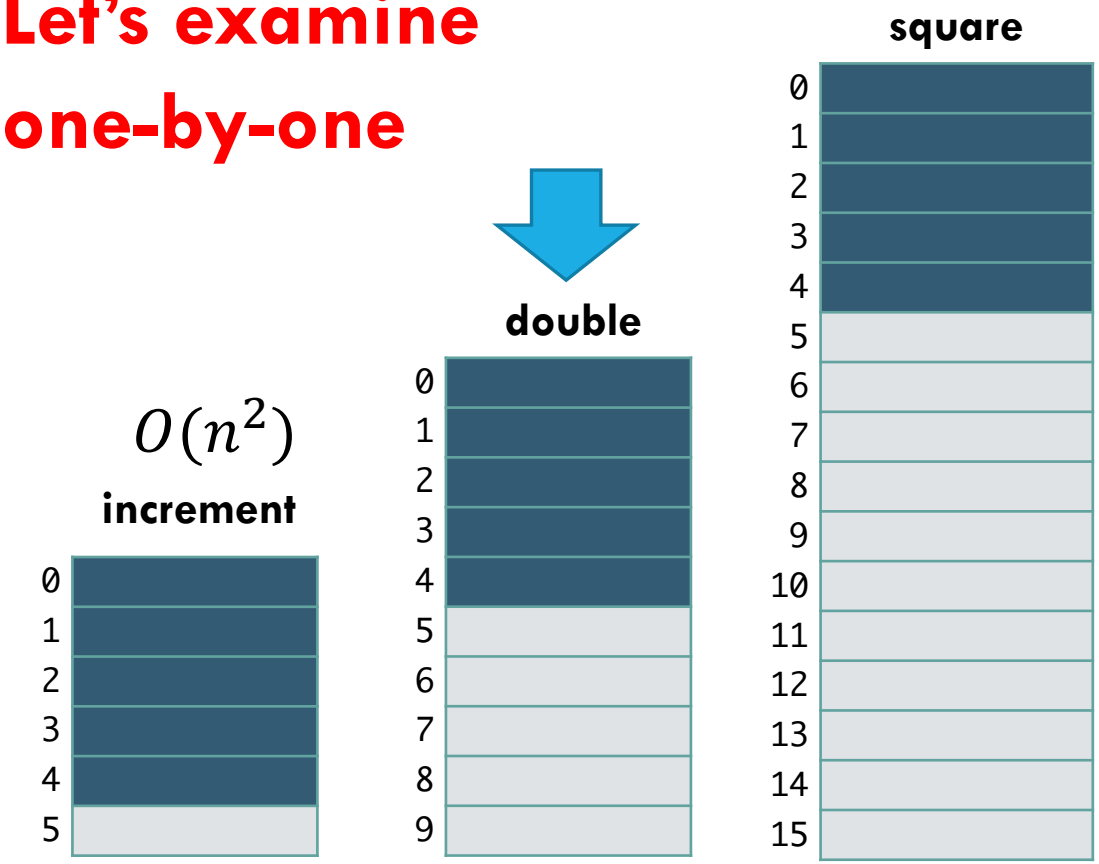
- If $n < m$ (table is not full), then $c_i = 1$
- If $n = m$ (table is full), then expand so, $c_i = i + (i-1) + 1 = 2i$
- Let's say m_0 is the initial size of the table

Total cost for inserting n elements: $O(n^2)$

Amortized cost for insert: $O(n)$

GROWING THE TABLE

Let's examine one-by-one



Which do you think is the better technique?

A. increment by 1
B. double the size
C. square the size
D. They are all the same
E. Definitely not the same, but I don't know which one.

Hint: You'll probably need:
$\sum_{k=0}^{\lfloor \log n \rfloor} 2^k \leq 2n$

DOUBLE THE SIZE

What is c_i (the cost of the i^{th} insert)?

- If $n < m$ (table is not full), then $c_i = 1$
- If $n = m$ (table is full), then expand so, $c_i = 2i + (i-1) + 1 = 3i$
  - Table is full only when i is an exact power of 2, i.e., $i = 2^j$ for some j
- Let's say m_0 is the initial size of the table

Hint: You'll probably need:
$\sum_{k=0}^{\lfloor \log n \rfloor} 2^k \leq 2n$

DOUBLE THE SIZE

What is c_i (the cost of the i^{th} insert)?

- If $n < m$ (table is not full), then $c_i = 1$
- If $n = m$ (table is full), then expand so, $c_i = 2i + (i-1) + 1 = 3i$
  - Table is full only when i is an exact power of 2, i.e., $i = 2^j$ for some j
- Let's say m_0 is the initial size of the table

What is the Total Cost to insert n elements where $n > m_0$?

$O(n)$.

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \log n \rfloor} c_{2^j}$$

$$\leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 3\,(2^j)$$

$$\leq n + 3\sum_{j=0}^{\lfloor \log n \rfloor} 2^j$$

$$< n + 6n = O(n)$$

Hint: You'll probably need: $\sum_{k=0}^{\lfloor \log n \rfloor} 2^k \leq 2n$

DOUBLE THE SIZE

What is c_i (the cost of the i^{th} insert)?

- If $n < m$ (table is not full), then $c_i = 1$
- If $n = m$ (table is full), then expand so, $c_i = 2i + (i-1) + 1 = 3i$
  - Table is full only when i is an exact power of 2, i.e., $m = 2^j$ for some j
- Let's say m_0 is the initial size of the table

What is the <u>average</u> (*amortized*) cost per insert over the sequence?

$O(1)$. Why?

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{m_0-1} c_i + \sum_{j=m_0}^{\lfloor \log n \rfloor} c_j$$

$$\leq m_0 + \sum_{j=0}^{\lfloor \log n \rfloor} 3\,(2^j)$$

$$\leq m_0 + 3\sum_{j=0}^{\lfloor \log n \rfloor} 2^j$$

$$< n + 6n = O(n)$$

Avg cost: $\frac{1}{n}\sum_{i=1}^{n} c_i \leq \frac{1}{n}(7n) = 7 = O(1)$

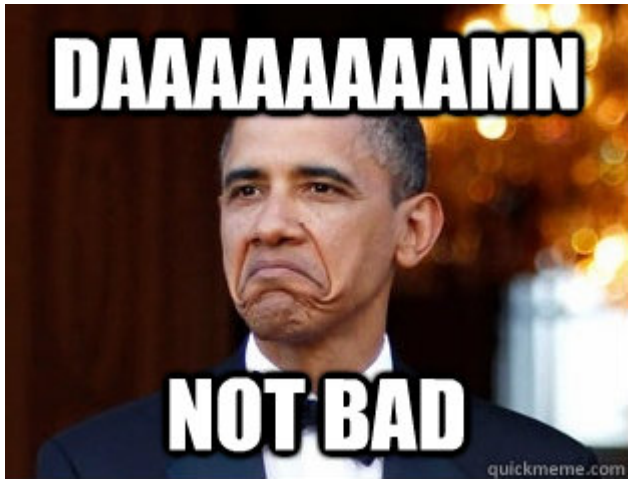
DOUBLE THE SIZE

What is c_i (the cost of the i^{th} insert)?

- If $n < m$ (table is not full), then $c_i = 1$
- If $n = m$ (table is full), then expand so, $c_i = 2i + (i-1) + 1 = 3i$
  - Table is full only when i is an exact power of 2, i.e., $m = 2^j$ for some j
- Let's say m_0 is the initial size of the table

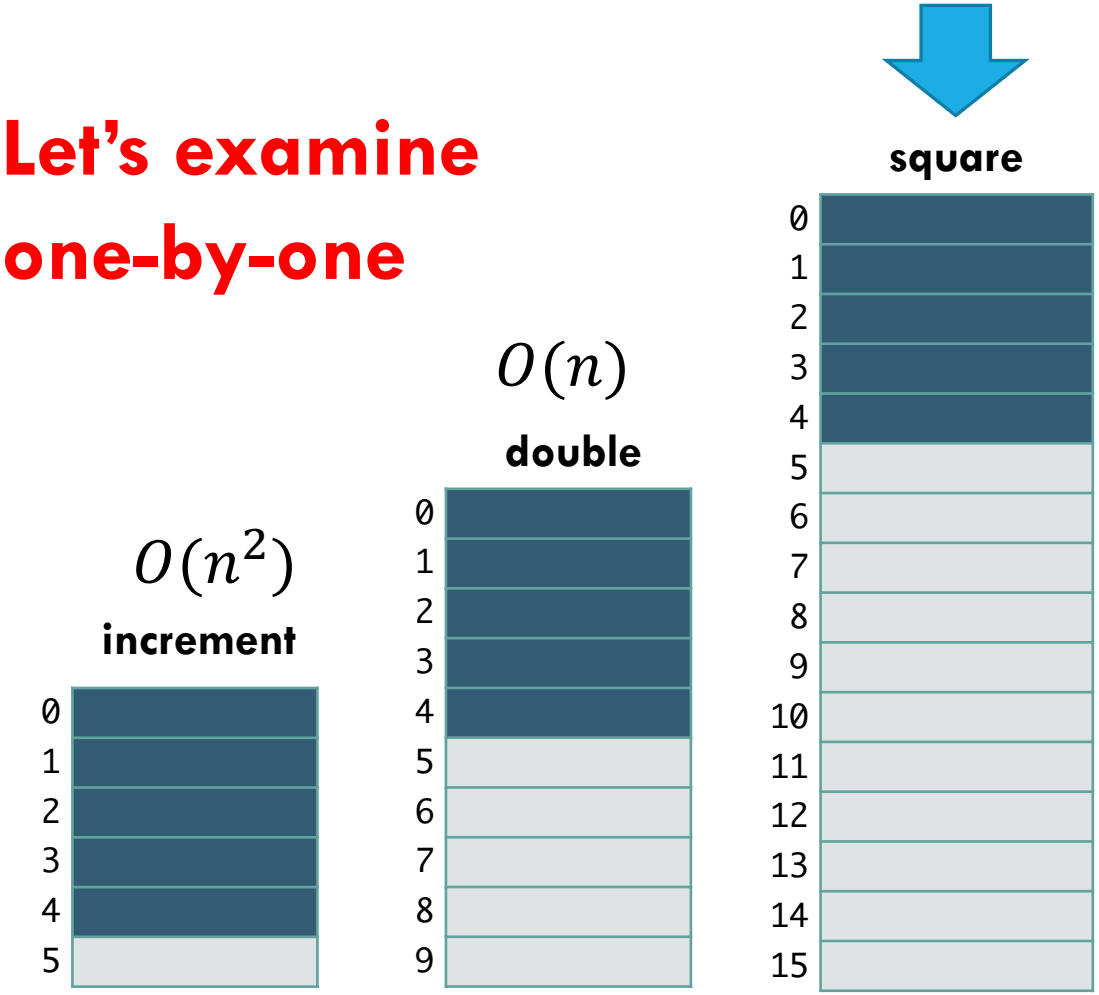
Total cost for inserting n elements: $O(n)$

Amortized cost for insert: $O(1)$



GROWING THE TABLE

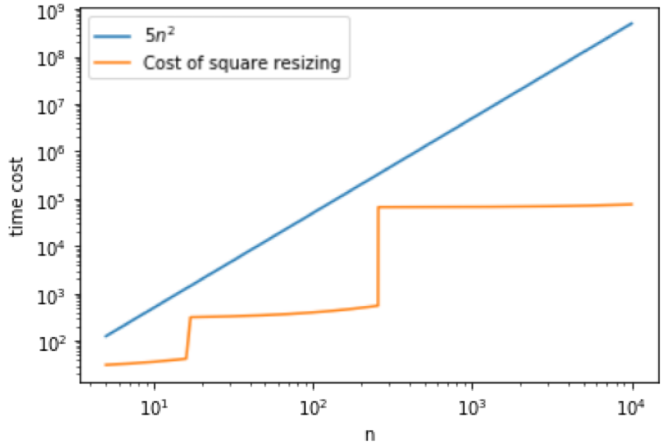
Let's examine one-by-one



Which do you think is the better technique?

A. increment by 1
B. double the size
C. square the size
D. They are all the same
E. Definitely not the same, but I don't know which one.

SQUARE THE SIZE



What is c_i (the cost of the i^{th} insert)?

- If $n < m$ (table is not full), then $c_i = 1$
- If $n = m$ (table is full), then expand so, $c_i = i^2 + (i-1) + 1 = i^2 + i$
  - Table is full only when $i = 2^{2^j}$ for some j
- Let's say m_0 is the initial size of the table

What is the Total Cost to insert n elements where $n > m_0$?

$$O(n^2).$$

$$\begin{aligned}
\sum_{i=1}^{n} c_i &\le n + \sum_{j=0}^{\log\log n} c_{2^{2^j}} \\
&\le n + \sum_{j=0}^{\log\log n} \left(2^{2^j}\right)^2 + 2^{2^j} \\
&\le n + \sum_{j=0}^{\log\log n} 2^{2^{j+1}} + 2^{2^j} \\
&= O(n^2)
\end{aligned}$$

SQUARE THE SIZE

What is c_i (the cost of the i^{th} insert)?

- If $n < m$ (table is not full), then $c_i = 1$
- If $n = m$ (table is full), then expand so, $c_i = i^2 + (i-1) + 1 = i^2 + i$
  - Table is full only when $i = 2^{2^j}$ for some j
- Let's say m_0 is the initial size of the table

Total cost for inserting n elements: $O(n^2)$

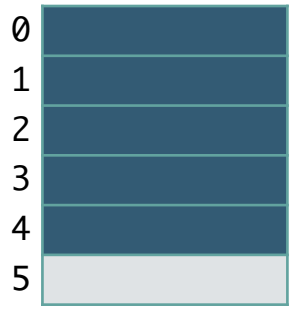
Amortized cost for insert: $O(n)$

GROWING THE TABLE

Let's examine one-by-one

$O(n^2)$

increment



$O(n)$

double

$O(n^2)$

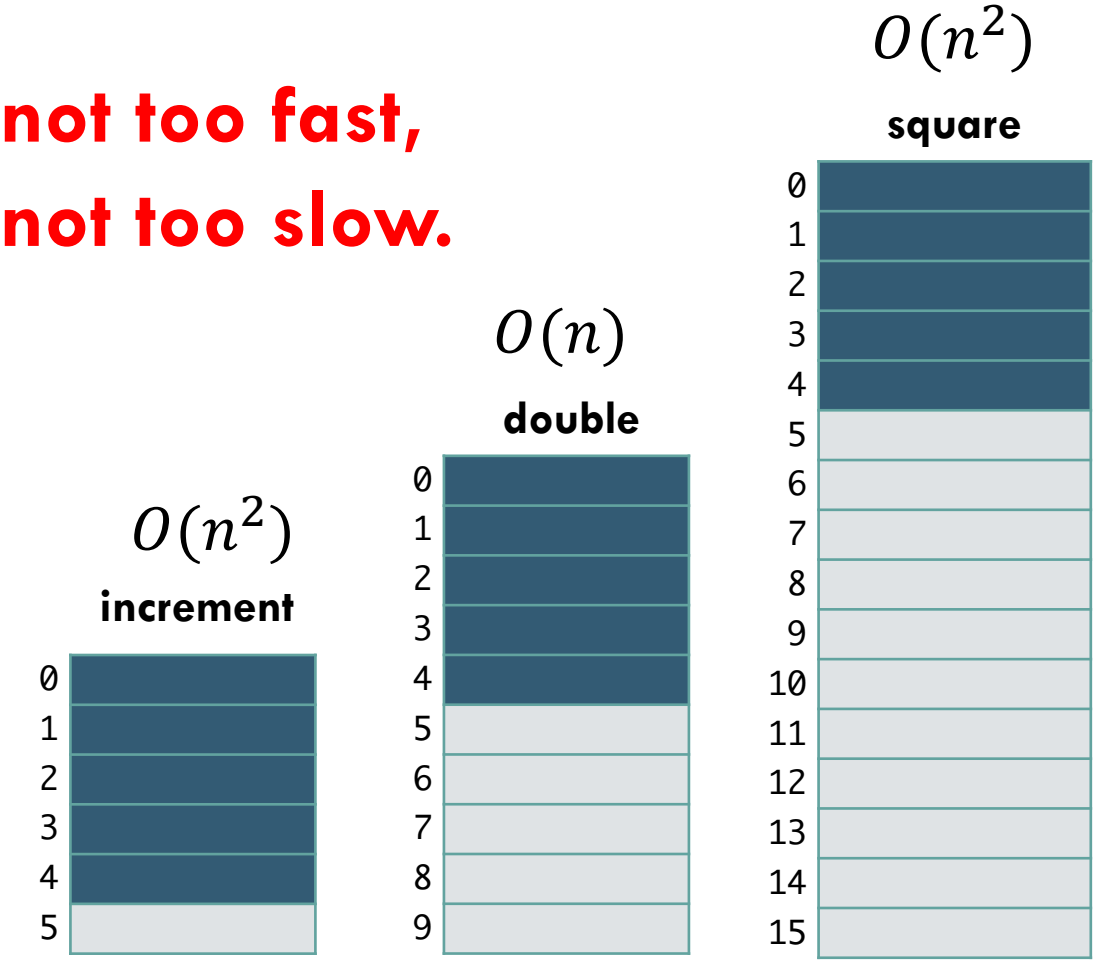
square

Which do you think is the better technique?

A. increment by 1
B. double the size
C. square the size
D. They are all the same
E. Definitely not the same, but I don't know which one.

GROWING THE TABLE

not too fast,
not too slow.



Which do you think is the better technique?

A. increment by 1
B. double the size
C. square the size
D. They are all the same
E. Definitely not the same, but I don't know which one.

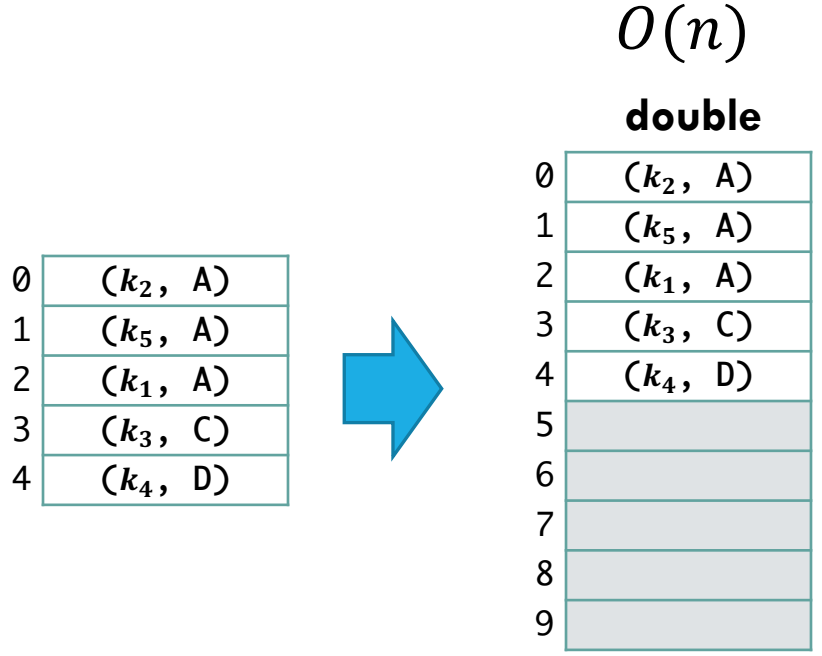
COSTS OF INSERTS

If we include cost of table resizing:

Assume: size doubling strategy

Cost of resize: $O(n)$

Worst case cost per insert is _____



COSTS OF INSERTS

If we include cost of table resizing:

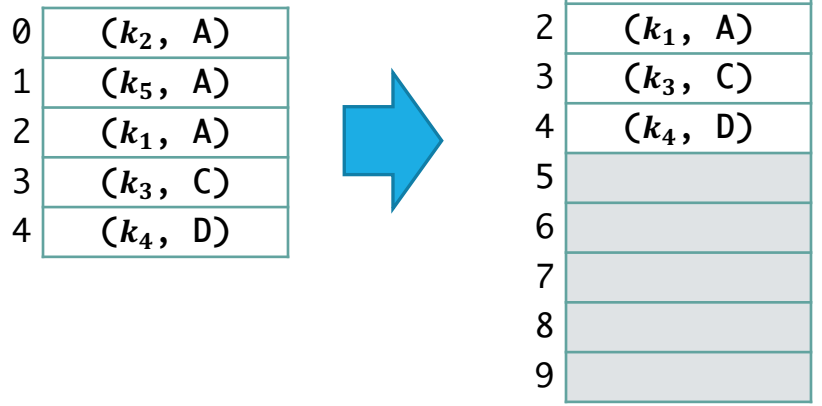
Assume: size doubling strategy

Cost of resize: $O(n)$

Worst case cost per insert is $O(n)$

(At each insert, there is a chance of resizing)

But average (amortized) cost over a sequence is $\boldsymbol{O(1)}$



AMORTIZED ANALYSIS

Technique for analyzing "average" cost

- Focus on overall cost of a *sequence* of operations (e.g., n inserts)
- Does not specify the cost of *a specific* operation (the 5th insert)

Common in data structure analysis

3 techniques:

- Aggregate Analysis (we just did this!)
- Accounting (or Banker's) Method
- Potential (or Physicist's) Method

AMORTIZED ANALYSIS: ACCOUNTING METHOD

Analogy: paying rent

- You don't pay rent every day
- S$900/month = $30/day

Definition:

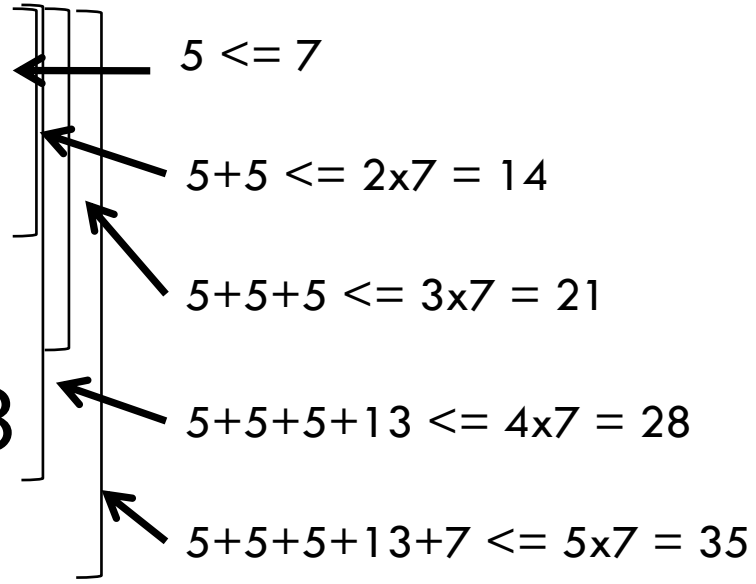
Operation has **amortized cost** $T(n)$ if for every integer k, the cost of k operations is $\leq kT(n)$

AMORTIZED ANALYSIS

Operation has **amortized cost** $T(n)$ if for every integer k, the cost of k operations is $\leq kT(n)$

Example: Amortized cost = 7

- insert: 5
- insert: 5
- insert: 5
- insert: 13
- insert: 7

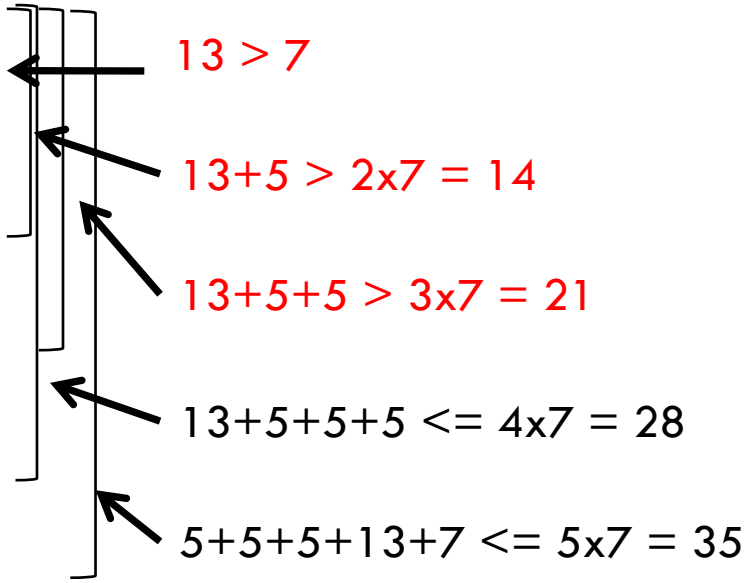


AMORTIZED ANALYSIS

Operation has **amortized cost** $T(n)$ if for every integer k, the cost of k operations is $\leq kT(n)$

Example: Amortized cost = 7 ? **NO!**

- insert: 13
- insert: 5
- insert: 5
- insert: 5
- insert: 7



AVERAGE COST OF INSERTS

Operation has **amortized cost** $T(n)$ if for every integer k, the cost of k operations is $\leq kT(n)$

Worst case insertion is $O(n)$

But: the insert operation has ***amortized*** cost $O(1)$!

ACCOUNTING METHOD

Operation has **amortized cost** $T(n)$ if for every integer k, the cost of k operations is $\leq kT(n)$

Analogy: Paying Rent

Imagine: you have some bank account B

Each operation adds $ to the bank account

Every step of the algorithm spends money

- Immediate money: to perform the operation
- Deferred money: from the bank account

ACCOUNTING METHOD: EXAMPLE

Bank account
$2

0	null
1	null
2	(k_1, A)
3	(k_2, B)
4	null

Each Hash Table has a bank account

Strategy:

- Select a charge amount for each insert.
- Ensure bank account is always big enough to pay for resizing.

Let:

- Actual cost per insert (without resizing) be c_i
- Amortized cost (charge) be $\widehat{c_i}$
- If $\widehat{c_i} > c_i$, there is some extra credit left in the account
  - $credit_i = \widehat{c_i} - c_i$
  - Can be used by future operations (e.g. resizing).
  - Total credit should never be negative!

ACCOUNTING METHOD: EXAMPLE

Bank account
$3

0	null
1	null
2	(k_1, A)
3	(k_2, B)
4	(k_3, C)

Each Hash Table has a bank account

Each time an element is added:

- $\widehat{c_i}$ =$2
- c_i =$1

A table with k new elements since last resize has k dollars in bank.

ACCOUNTING METHOD: EXAMPLE

Bank account
$5

0	(k_4, D)
1	(k_5, E)
2	(k_1, A)
3	(k_2, B)
4	(k_3, C)

Each Hash Table has a bank account

When 5th element is added:

- $\widehat{c_i}$ =\$2
- c_i =\$1 + \$5

A table with k new elements since last resize has k dollars in bank.

ACCOUNTING METHOD: EXAMPLE

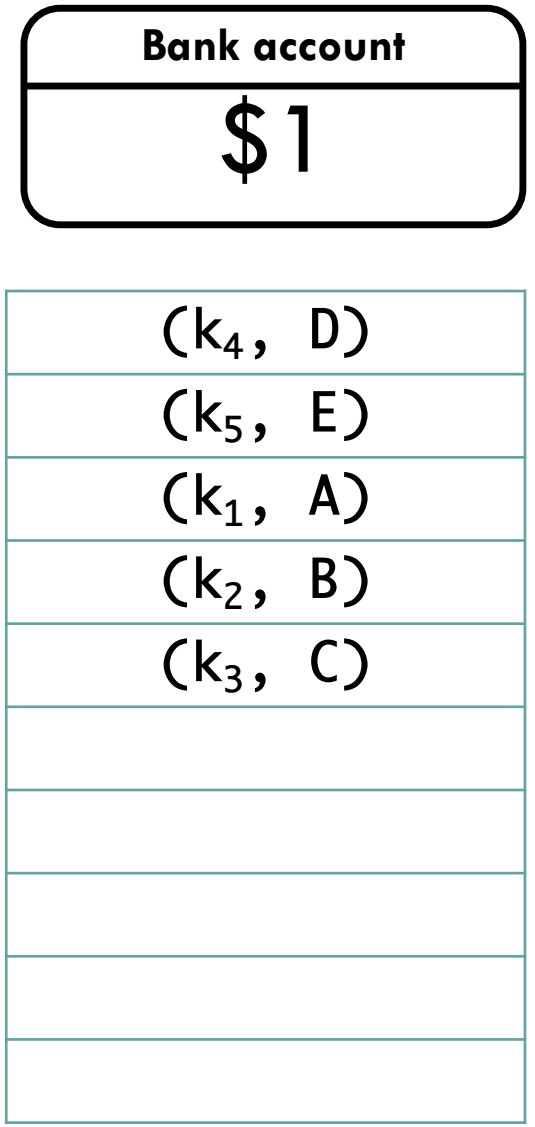
Each Hash Table has a bank account

When 5th element is added:

- $\widehat{c_i}$ =\$2
- c_i =\$1 + \$5

A table with k new elements since last resize has k dollars in bank.

Pay for resize!



ACCOUNTING METHOD: EXAMPLE

Bank account
$1

0	(k_4, D)
1	(k_5, E)
2	(k_1, A)
3	(k_2, B)
4	(k_3, C)
5	
6	
7	
8	
9	

Total Cost: Inserting k elements costs:

- **Deferred dollars:** $O(k)$ [pay for resizing]
- **Immediate dollars:** $O(k)$ [for inserting elements in table]
- Total = Deferred + Immediate = $O(k)$

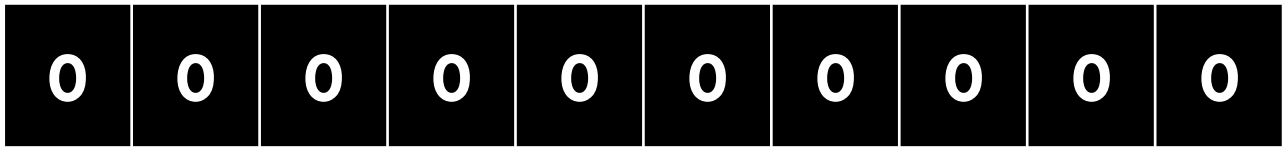
Average cost per operation:

- Deferred dollars: $O(1)$
- Immediate dollars: $O(1)$
- **Total = $O(1)$ per operation**

EXAMPLE: BINARY COUNTER

Counter ADT:

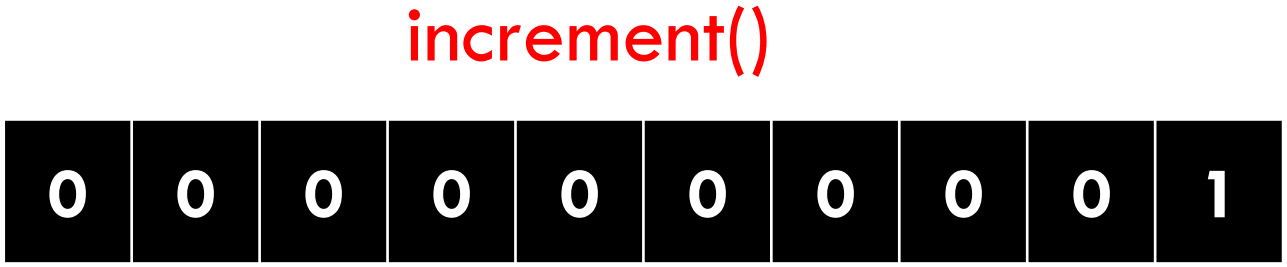
- increment()
- read()



EXAMPLE: BINARY COUNTER

Counter ADT:

- increment()
- read()

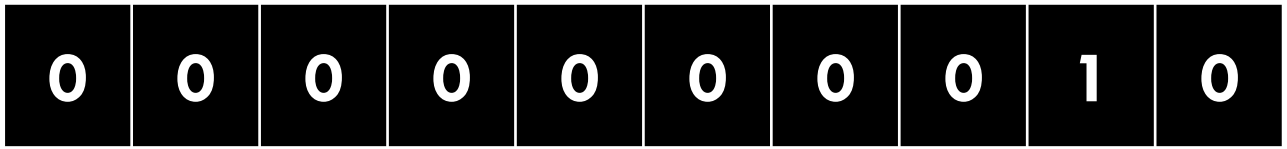


EXAMPLE: BINARY COUNTER

Counter ADT:

- increment()
- read()

increment(), increment()

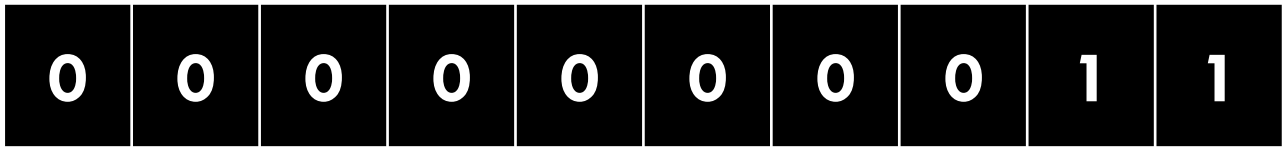


EXAMPLE: BINARY COUNTER

Counter ADT:

- increment()
- read()

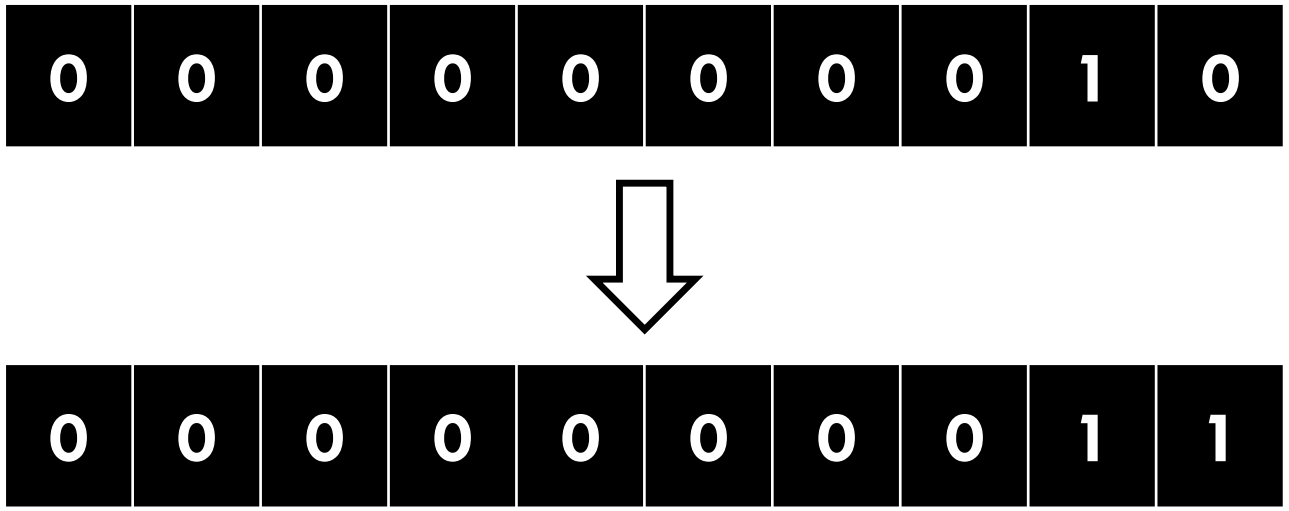
increment(), increment(), increment()



WORST CASE COST OF INCREMENTS

Poll Everywhere

https://bit.ly/2LvG9bq



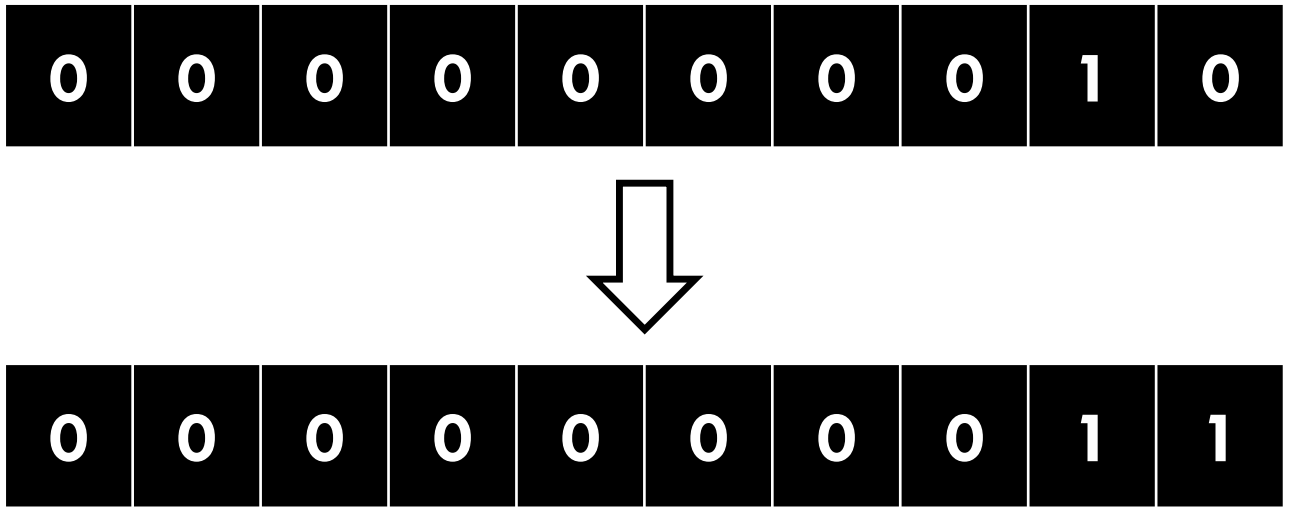
What is the worst-case cost of incrementing the counter if the max value is n?

A. $O(1)$
B. $O(\log n)$
C. $O(n)$
D. $O(n^2)$
E. I have no freaking idea man.

WORST CASE COST OF INCREMENTS

Poll Everywhere

https://bit.ly/2LvG9bq



What is the worst-case cost of incrementing the counter if the max value is n?

A. $O(1)$

B. $O(\log n)$

C. $O(n)$

D. $O(n^2)$

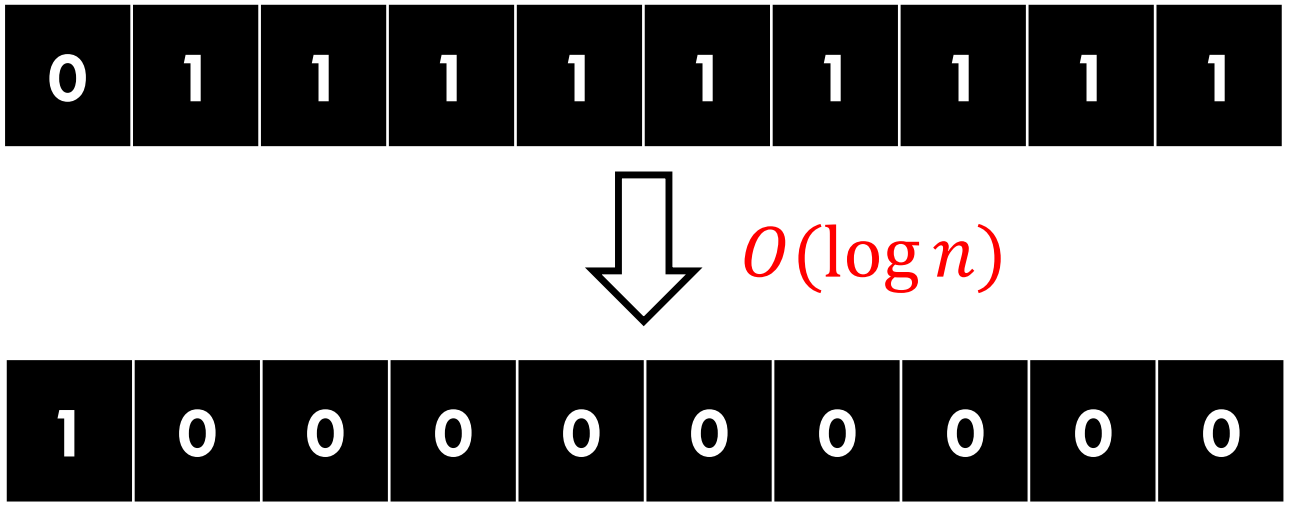
E. I have no freaking idea man.

EXAMPLE: BINARY COUNTER

Counter ADT:

- increment()
- read()

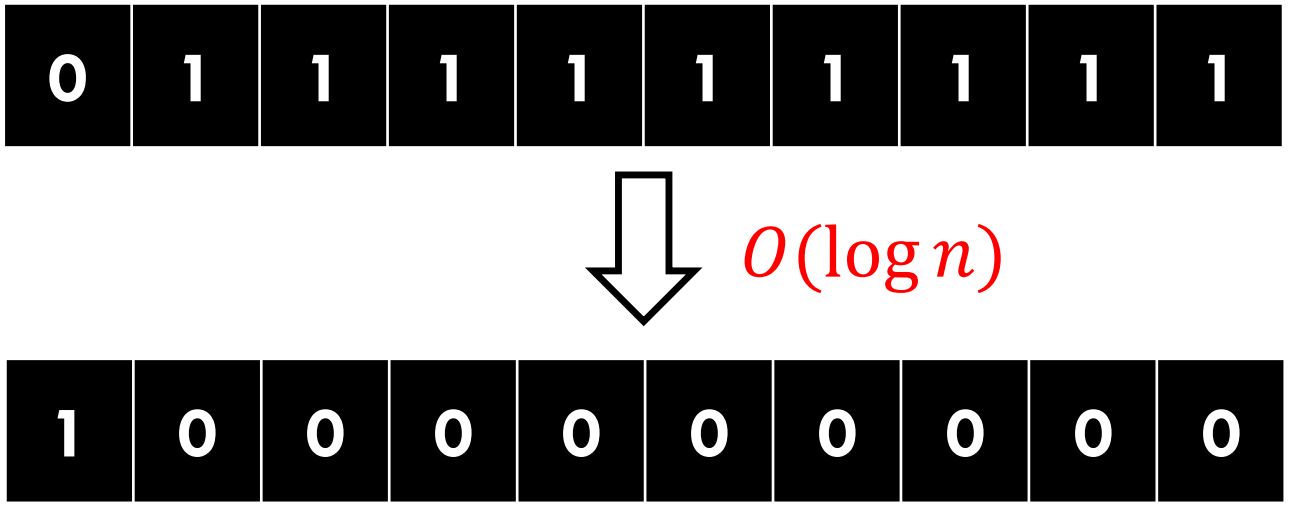
Some increments are expensive…



EXAMPLE: BINARY COUNTER

Question: If we increment the counter to n, what is the average cost per operation?

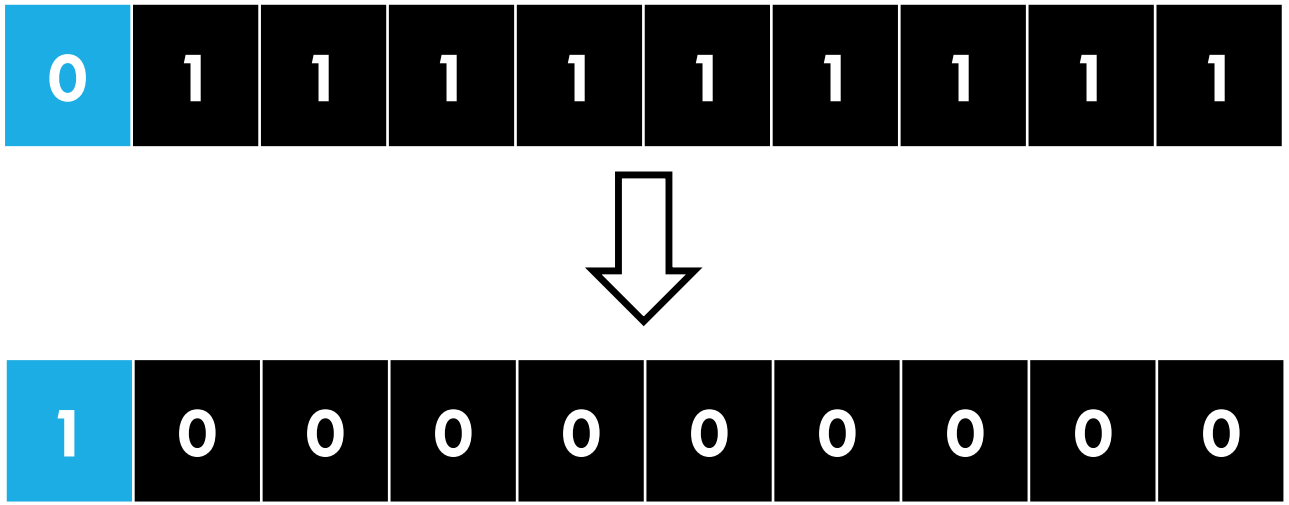
- Easy answer: $O(\log n)$
- Let's do a more careful analysis….



EXAMPLE: BINARY COUNTER

Observation:

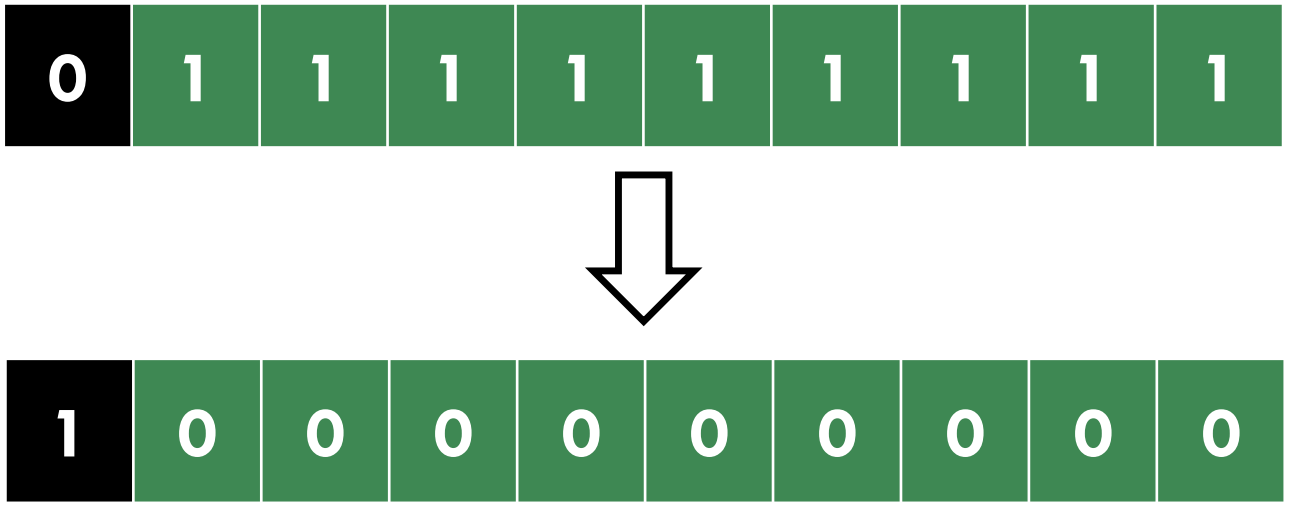
During each increment, only <u>one</u> bit is changed from: 0 → 1



EXAMPLE: BINARY COUNTER

Observation:

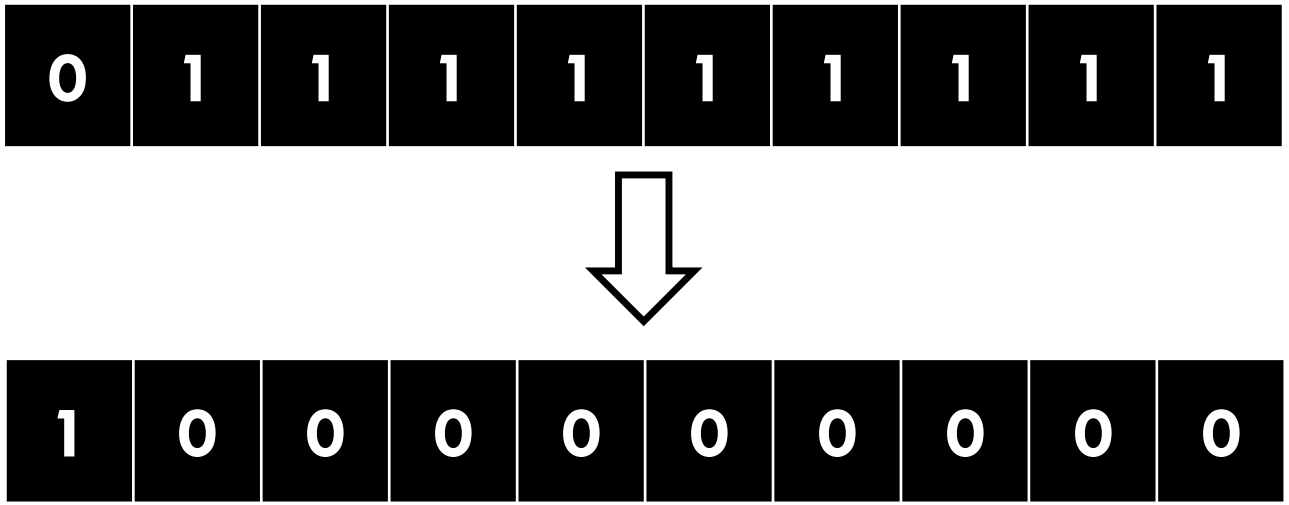
During each increment, <u>many</u> bits may be changed from: 1 → 0



EXAMPLE: BINARY COUNTER

Accounting method: each bit has a bank account.

Whenever you change it from 0→1, add one dollar.

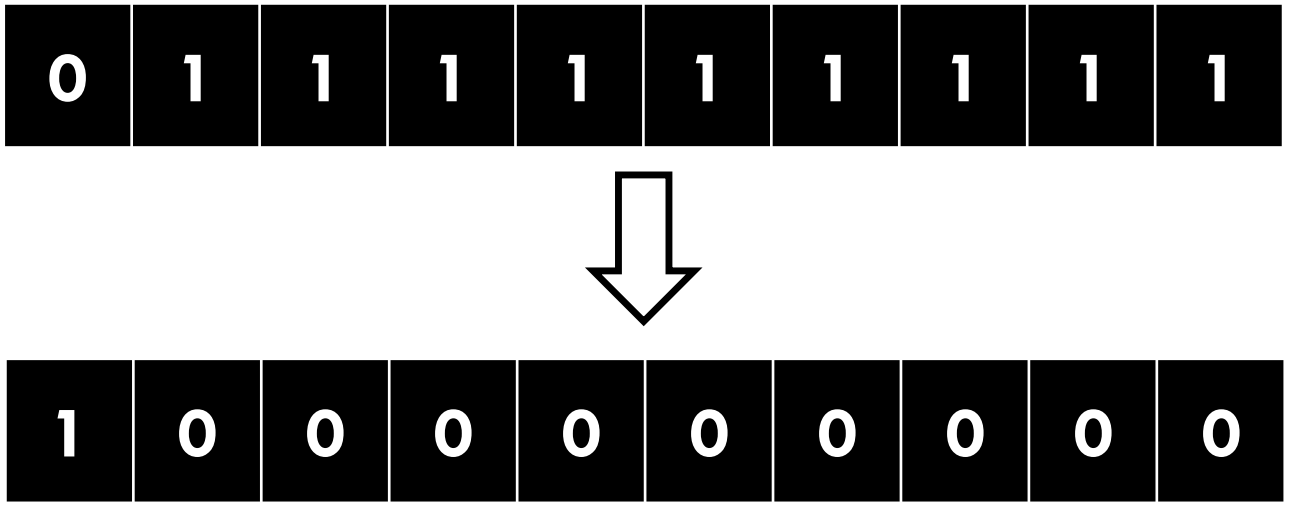


EXAMPLE: BINARY COUNTER

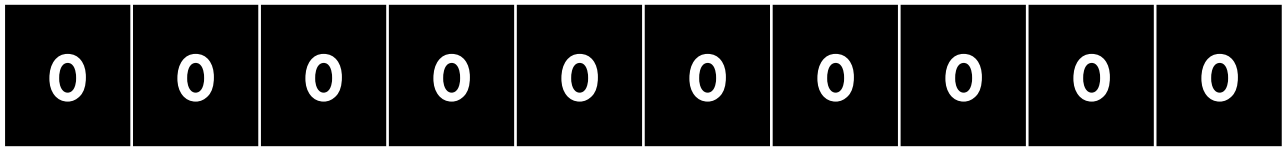
Accounting method: each bit has a bank account.

Whenever you change it from 0→1, add one dollar.

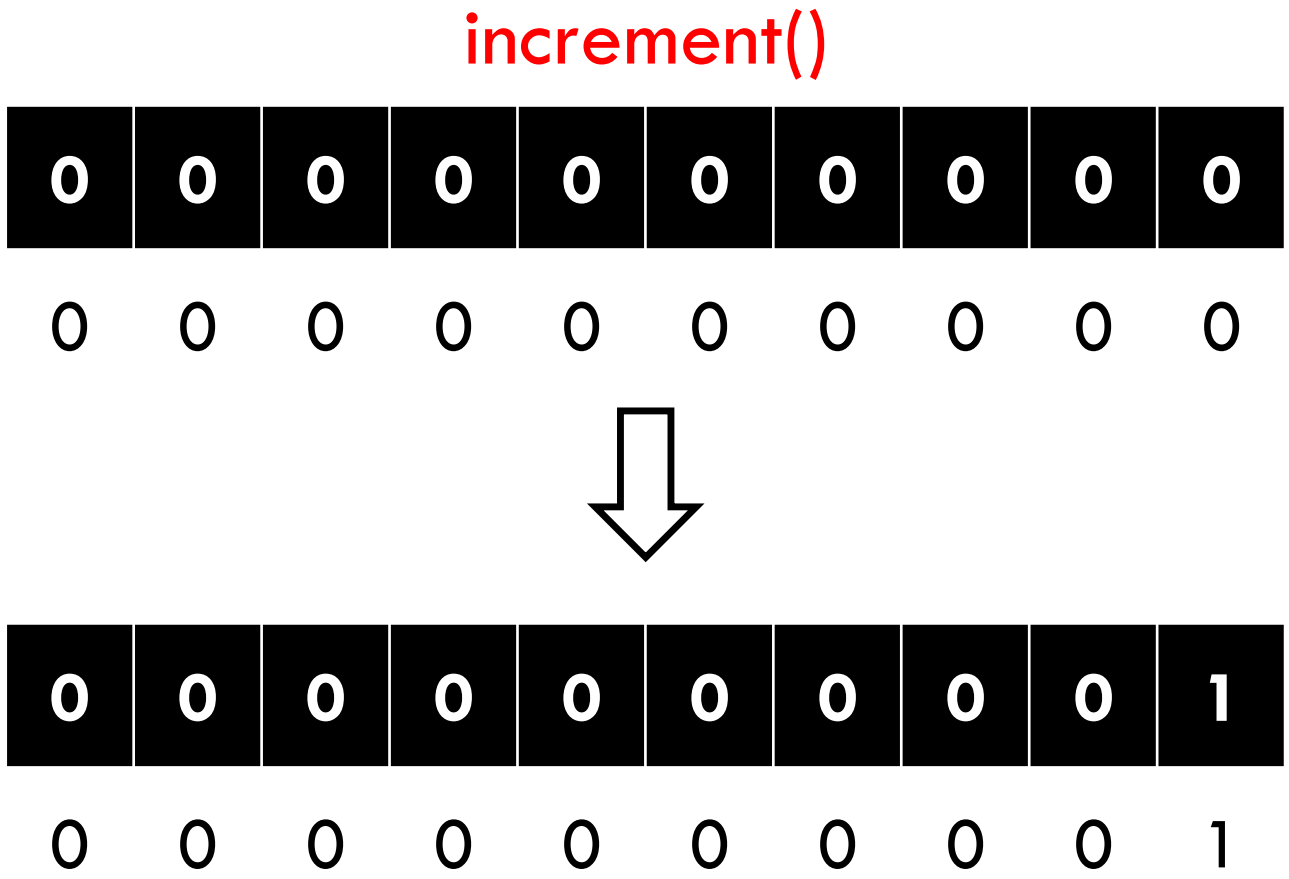
Whenever you change it from 1→0, pay one dollar.



EXAMPLE: BINARY COUNTER

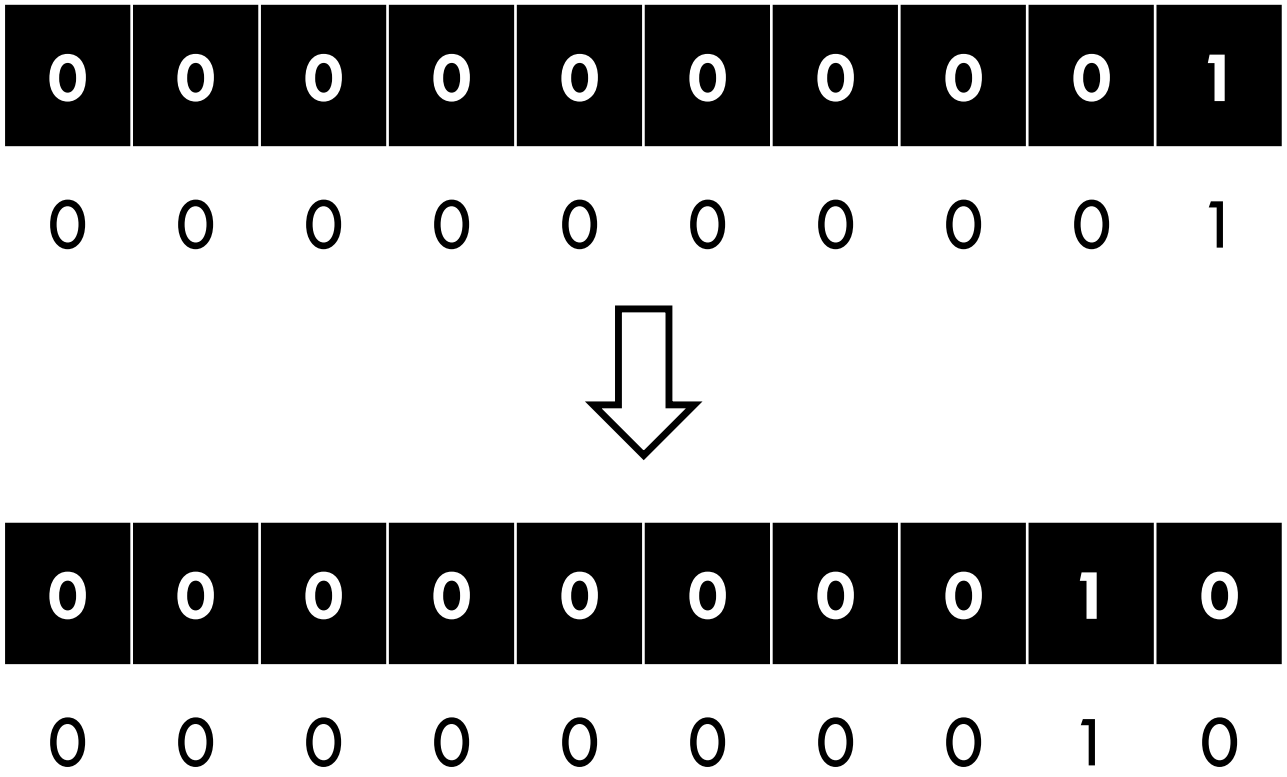


EXAMPLE: BINARY COUNTER



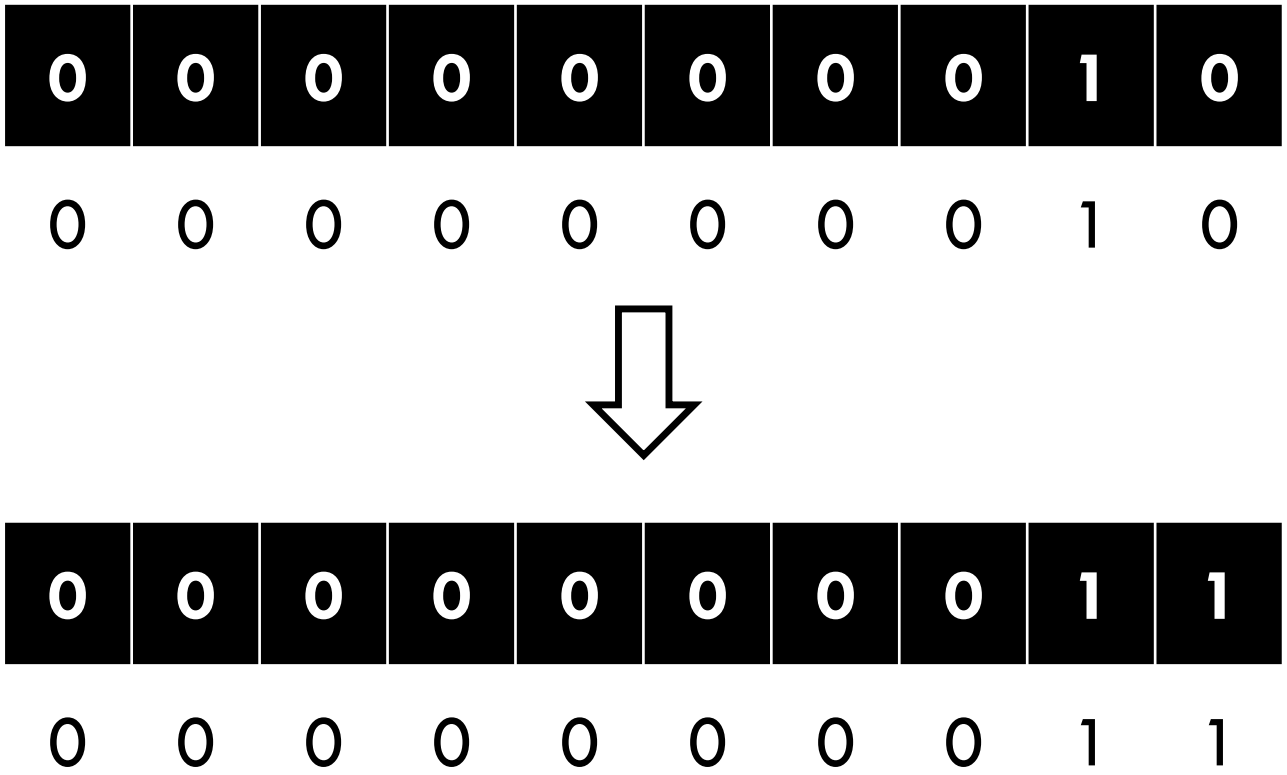
EXAMPLE: BINARY COUNTER

increment(), increment()

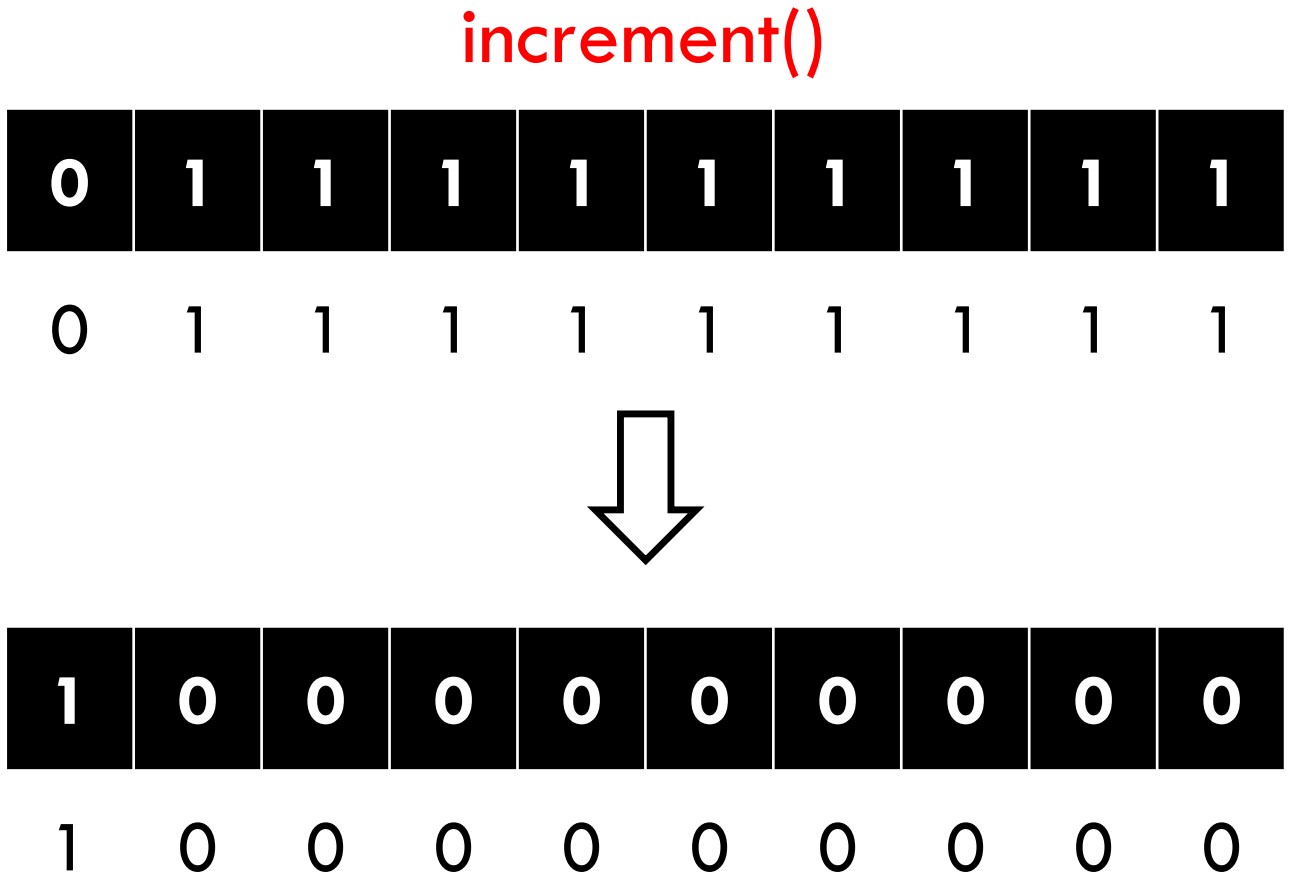


EXAMPLE: BINARY COUNTER

increment(), increment(), increment()



EXAMPLE: BINARY COUNTER

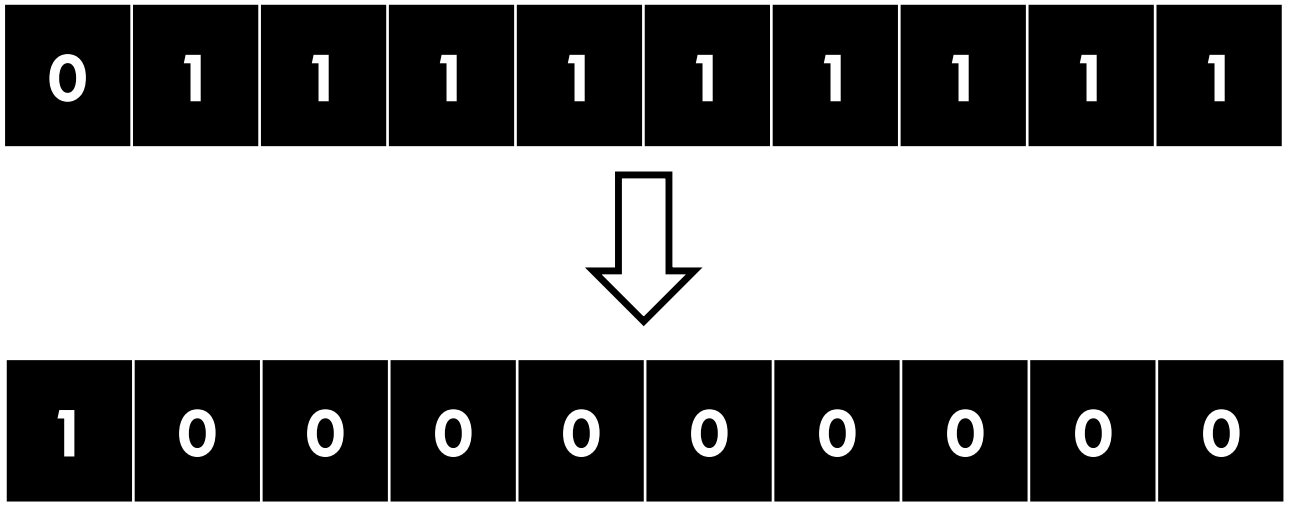


EXAMPLE: BINARY COUNTER

Amortized cost of increment: $2 = \boldsymbol{O(1)}$

- One dollar to switch <u>one</u> 0→1
- One dollar (for bank account of switched bit).

(All switches from 1→0 paid for by bank account.)



AMORTIZED ANALYSIS

Operation has **amortized cost** $T(n)$ if for every integer k, the cost of k operations is $\leq kT(n)$

Just a little taste!

Intuitive idea of the approach

More fun in CS3230!

Back to Hash Tables …

CUCKOO HASHING

Did you know?

- Cuckoos lay eggs in other birds nests.
- When the cuckoo bird hatches, it pushes eggs/chicks out of the nest.

What a neat idea!

- Open addressing policy!
- Described by Rasmus Pagh and Flemming Friche Rodler in 2001.

CUCKOO HASHING: HOW DOES IT WORK?

Use 2 hash functions $h_1(k)$ and $h_2(k)$

So, each key only has **2 possible locations**

0	
1	k_1
2	
3	k_2
4	k_3
5	k_4
6	
7	
8	k_5
9	
10	

CUCKOO HASHING: HOW DOES IT WORK?

search(k_1)

$h_1(k_1) = 1$

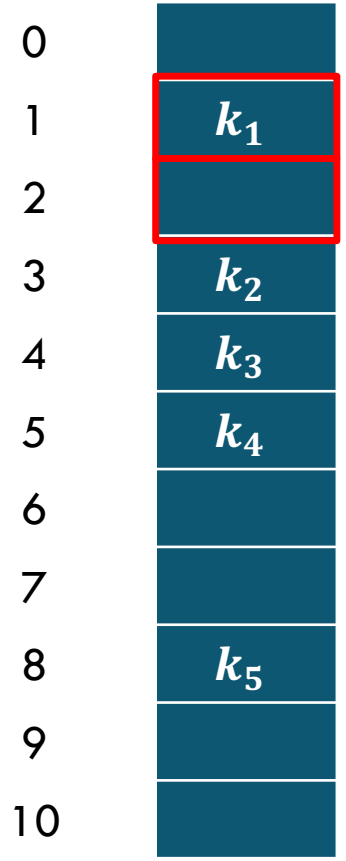
$h_2(k_1) = 2$

Use 2 hash functions $h_1(k)$ and $h_2(k)$

So, each key only has **2 possible locations**

Operations:

- **Lookup:** only check the 2 possible locations
  - $O(1)$ worst-case time!



CUCKOO HASHING: HOW DOES IT WORK?

Use 2 hash functions $h_1(k)$ and $h_2(k)$

So, each key only has **2 possible locations**

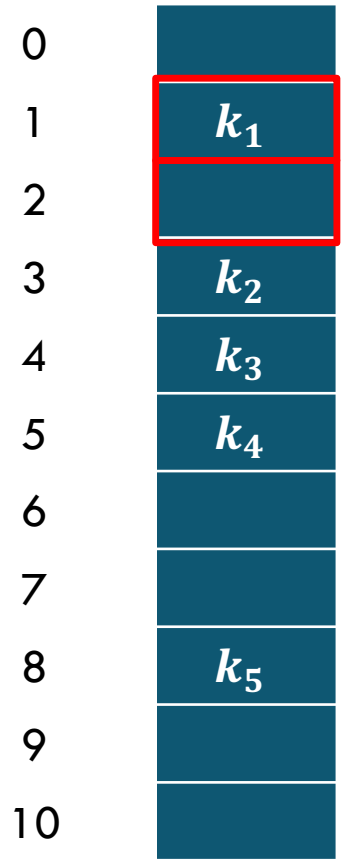
Operations:

- **Lookup:** only check the 2 possible locations
  - $O(1)$ worst-case time!
- **Deletion:** only check the 2 possible locations and delete accordingly.
  - Also $O(1)$ worst-case time

delete(k_1)

$h_1(k_1) = 1$

$h_2(k_1) = 2$



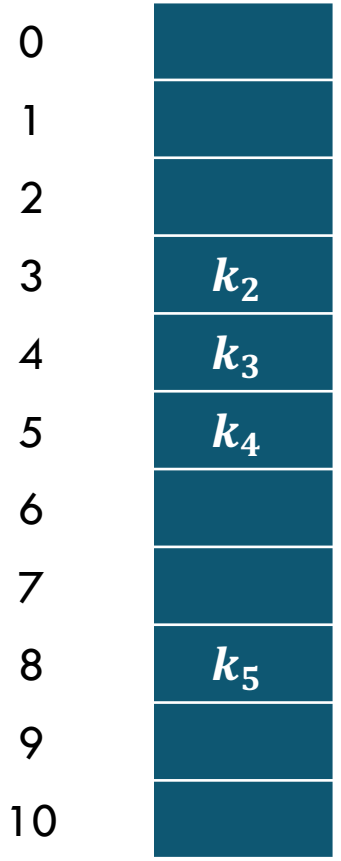
CUCKOO HASHING: HOW DOES IT WORK?

Use 2 hash functions $h_1(k)$ and $h_2(k)$

So, each key only has **2 possible locations**

Operations:

- **Lookup:** only check the 2 possible locations
  - $O(1)$ worst-case time!
- **Deletion:** only check the 2 possible locations and delete accordingly.
  - Also $O(1)$ worst-case time



delete(k_1)

$h_1(k_1) = 1$

$h_2(k_1) = 2$

CUCKOO HASHING: INSERTIONS

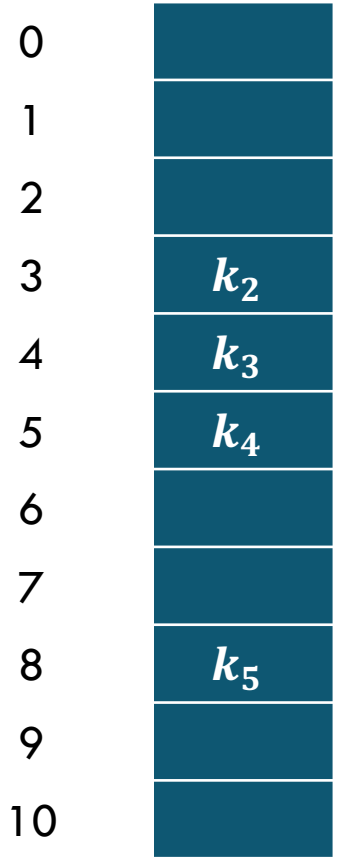
`insert`(k_1)

$h_1(k_1) = 1$

Idea: When inserting, if bucket is taken, push existing key out!

Steps:

- Set $p = h_1(k)$
- Repeat n times
  - If $A[p]$ is empty then
    - $A[p] = k$
    - return
  - $t = A[p]$; $A[p] = k$; $k = t$
  - If p $= h_1(k)$ then p $= h_2(k)$ else p $= h_1(k)$
- rehash(); insert(k)



CUCKOO HASHING: INSERTIONS

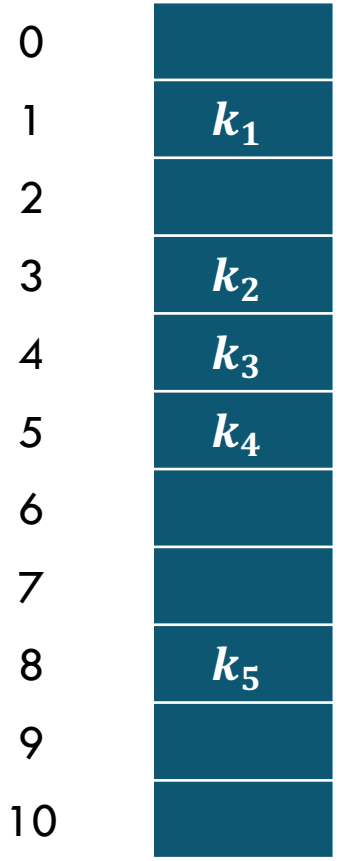
`insert`(k_1)

$h_1(k_1) = 1$

Idea: When inserting, if bucket is taken, push existing key out!

Steps:

- Set $p = h_1(k)$
- Repeat n times
  - If $A[p]$ is empty then
    - $A[p] = k$
    - return
  - $t = A[p]$; $A[p] = k$; $k = t$
  - If p $= h_1(k)$ then p $= h_2(k)$ else p $= h_1(k)$
- rehash(); insert(k)



CUCKOO HASHING: INSERTIONS

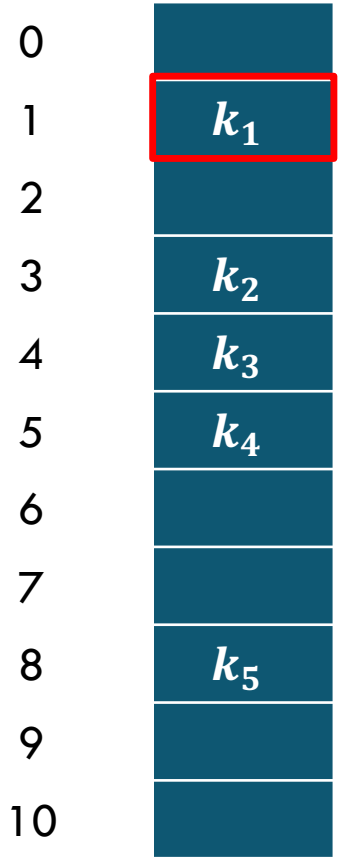
Idea: When inserting, if bucket is taken, push existing key out!

Steps:

- Set $p = h_1(k)$
- Repeat n times
  - If $A[p]$ is empty then
    - $A[p] = k$
    - return
  - $t = A[p]$; $A[p] = k$; $k = t$
  - If p $= h_1(k)$ then p $= h_2(k)$ else p $= h_1(k)$
- rehash(); insert(k)

insert(k_6)

$h_1(k_6) = 1$

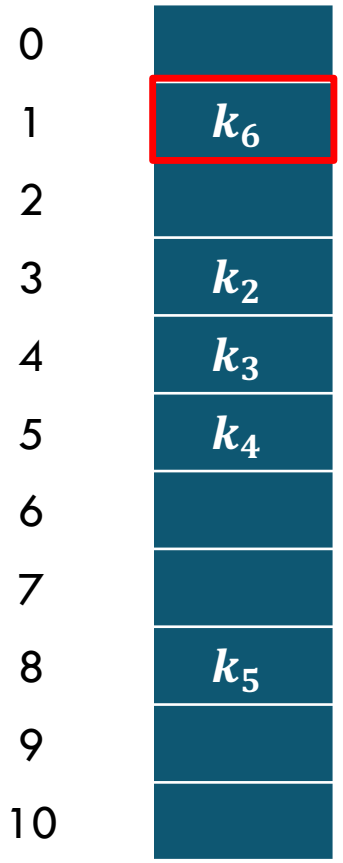


CUCKOO HASHING: INSERTIONS

Idea: When inserting, if bucket is taken, push existing key out!

Steps:

- Set $p = h_1(k)$
- Repeat n times
  - If $A[p]$ is empty then
    - $A[p] = k$
    - return
  - $t = A[p]$; $A[p] = k$; $k = t$
  - If p $= h_1(k)$ then p $= h_2(k)$ else p $= h_1(k)$
- rehash(); insert(k)



insert(k_6)

$h_1(k_6) = 1$

k_1 got pushed out!

$h_1(k_1) = 1$

$h_2(k_1) = 2$

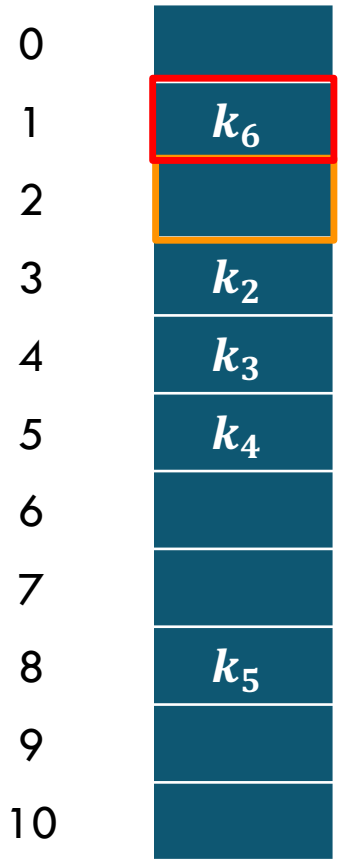
CUCKOO HASHING: INSERTIONS

insert(k_6)

Idea: When inserting, if bucket is taken, push existing key out!

Steps:

- Set $p = h_1(k)$
- Repeat n times
  - If $A[p]$ is empty then
    - $A[p] = k$
    - return
  - $t = A[p]$; $A[p] = k$; $k = t$
  - If p $= h_1(k)$ then p $= h_2(k)$ else p $= h_1(k)$
- rehash(); insert(k)



k_1 got pushed out!

$h_1(k_1) = 1$

$h_2(k_1) = 2$

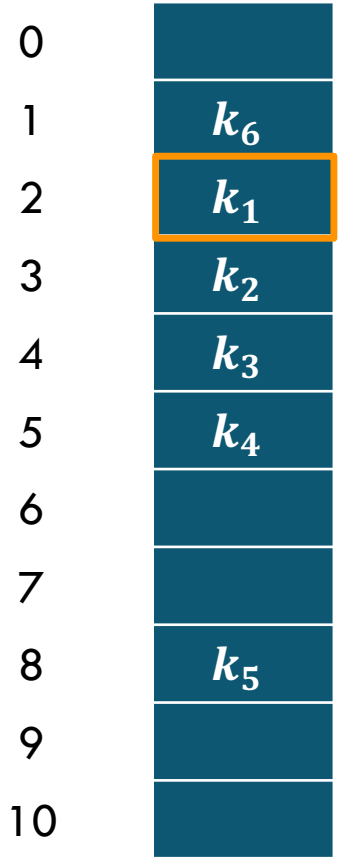
CUCKOO HASHING: INSERTIONS

`insert`(k_6)

Idea: When inserting, if bucket is taken, push existing key out!

Steps:

- Set $p = h_1(k)$
- Repeat n times
  - If $A[p]$ is empty then
    - $A[p] = k$
    - return
  - $t = A[p]$; $A[p] = k$; $k = t$
  - If p $= h_1(k)$ then p $= h_2(k)$ else p $= h_1(k)$
- rehash(); insert(k)



k_1 got pushed out!

$h_1(k_1) = 1$

$h_2(k_1) = 2$

CUCKOO HASHING: INSERTIONS

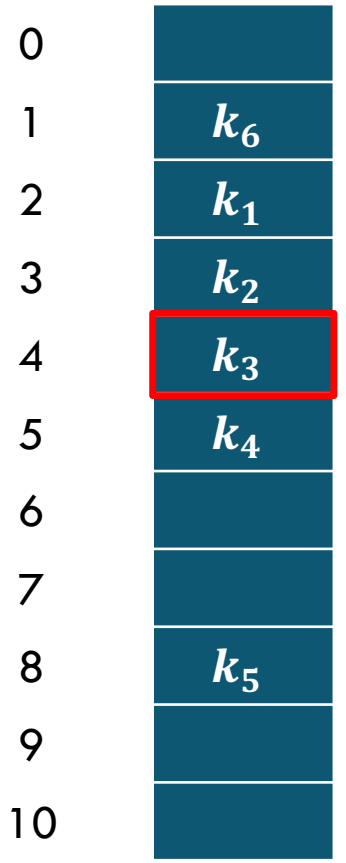
`insert`(k_7)

$h_1(k_7) = 4$

Idea: When inserting, if bucket is taken, push existing key out!

Steps:

- Set $p = h_1(k)$
- Repeat n times
  - If $A[p]$ is empty then
    - $A[p] = k$
    - return
  - $t = A[p]$; $A[p] = k$; $k = t$
  - If p $= h_1(k)$ then p $= h_2(k)$ else p $= h_1(k)$
- rehash(); insert(k)

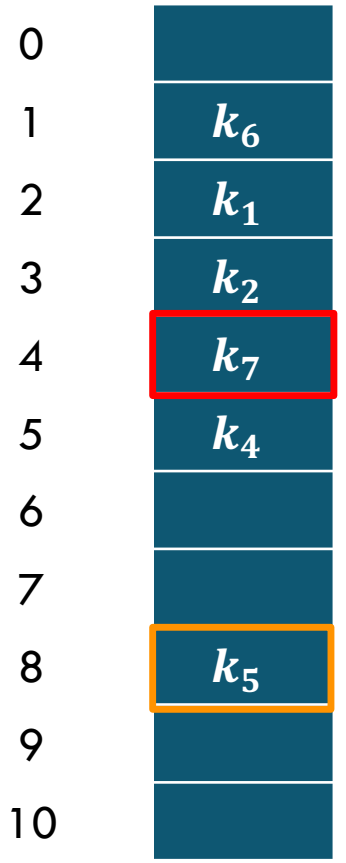


CUCKOO HASHING: INSERTIONS

Idea: When inserting, if bucket is taken, push existing key out!

Steps:

- Set $p = h_1(k)$
- Repeat n times
  - If $A[p]$ is empty then
    - $A[p] = k$
    - return
  - $t = A[p]$; $A[p] = k$; $k = t$
  - If p $= h_1(k)$ then p $= h_2(k)$ else p $= h_1(k)$
- rehash(); insert(k)



insert(k_7)

$h_1(k_7) = 4$

k_3 got pushed out!

$h_1(k_3) = 4$

$h_2(k_3) = 8$

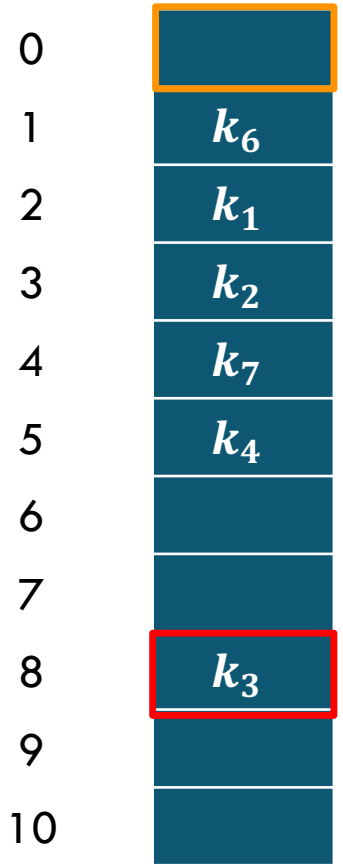
CUCKOO HASHING: INSERTIONS

insert(k_7)

Idea: When inserting, if bucket is taken, push existing key out!

Steps:

- Set $p = h_1(k)$
- Repeat n times
  - If $A[p]$ is empty then
    - $A[p] = k$
    - return
  - $t = A[p]$; $A[p] = k$; $k = t$
  - If p $= h_1(k)$ then p $= h_2(k)$ else p $= h_1(k)$
- rehash(); insert(k)



k_5 got pushed out!

$h_1(k_5) = 0$

$h_2(k_5) = 8$

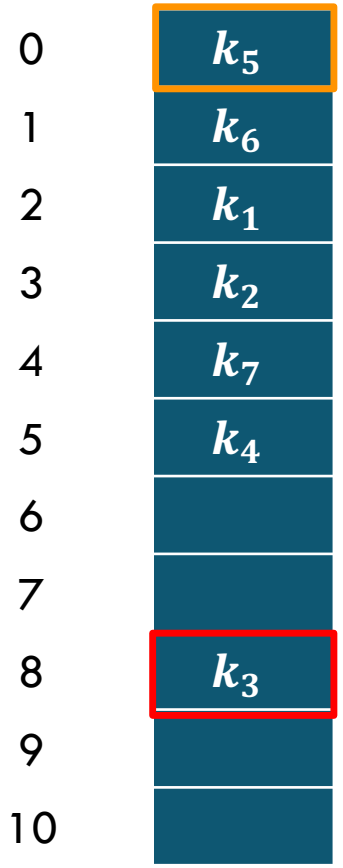
CUCKOO HASHING: INSERTIONS

insert(k_7)

Idea: When inserting, if bucket is taken, push existing key out!

Steps:

- Set $p = h_1(k)$
- Repeat n times
  - If $A[p]$ is empty then
    - $A[p] = k$
    - return
  - $t = A[p]$; $A[p] = k$; $k = t$
  - If p $= h_1(k)$ then p $= h_2(k)$ else p $= h_1(k)$
- rehash(); insert(k)



k_5 got pushed out!

$h_1(k_5) = 0$

$h_2(k_5) = 8$

CUCKOO HASHING: PERFORMANCE

Insertions seem to be quite complicated…

But: it takes expected $O(1)$ amortized time!

Analysis requires (a little) graph theory

To be continued in Week 12 …

PROBLEM: FIND ME A THIEF!

The Singapore Police wants some help:

They want to quickly look through a database of criminals based on fingerprints. Each lookup should be very fast.

Can you help?

Done! Use a Hash Table (assuming fingerprint detector is reliable)

PROBLEM: WHO IS ONLINE?

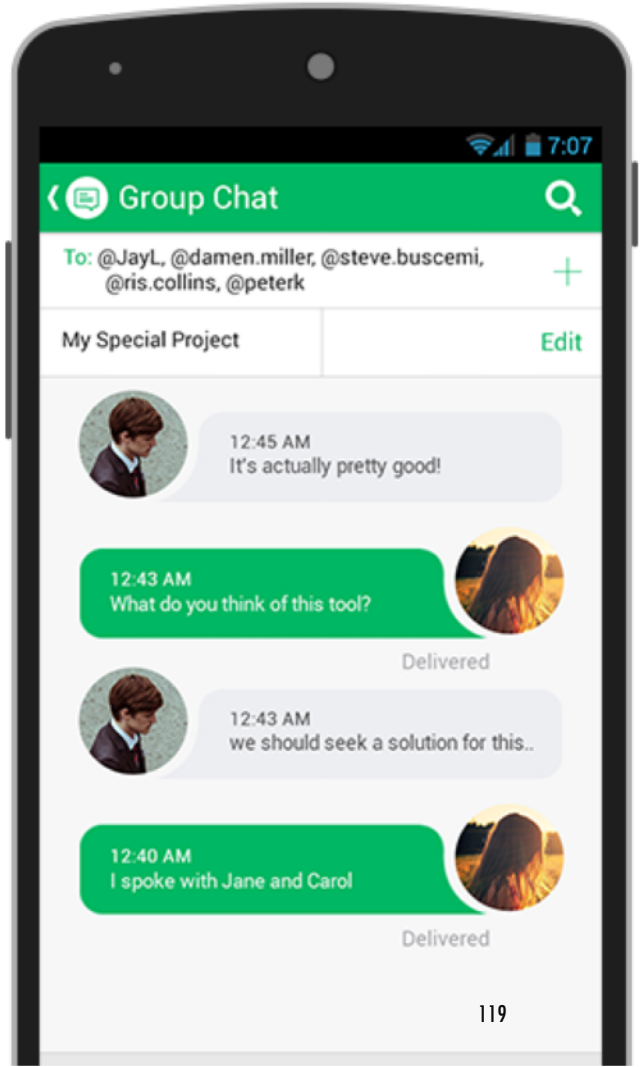
Social Media / Chat Applications

I have friend list:

- Ah Beng
- Sangheetha
- Charlie
- Mariam

How do I quickly check who is online?

Use a Hash Table!



SOME OTHER PROBLEMS…

Email black-list:

- spam@spam.com
- update@adobeflash.com **hash table!**
- harold@comp.nus.edu.sg

Representing Objects in Javascript **hash table!**

Patient records lookup **hash table!**

and so on …

SUMMARY

Symbol Tables are *pervasive*

Hash tables are fast, efficient data structures for implementing symbol tables

- Under optimistic assumption, provably so.
- In the real world, often so.
- But not perfect!

Beats BSTs:

- for searching.
- but: gave up “order” operations.

LEARNING OUTCOMES

By the end of this session, students should be able to:

- Describe the **division and multiplication methods** for generating hashes.
- Explain how to **resize (grow) hash tables dynamically**.

QUESTIONS?