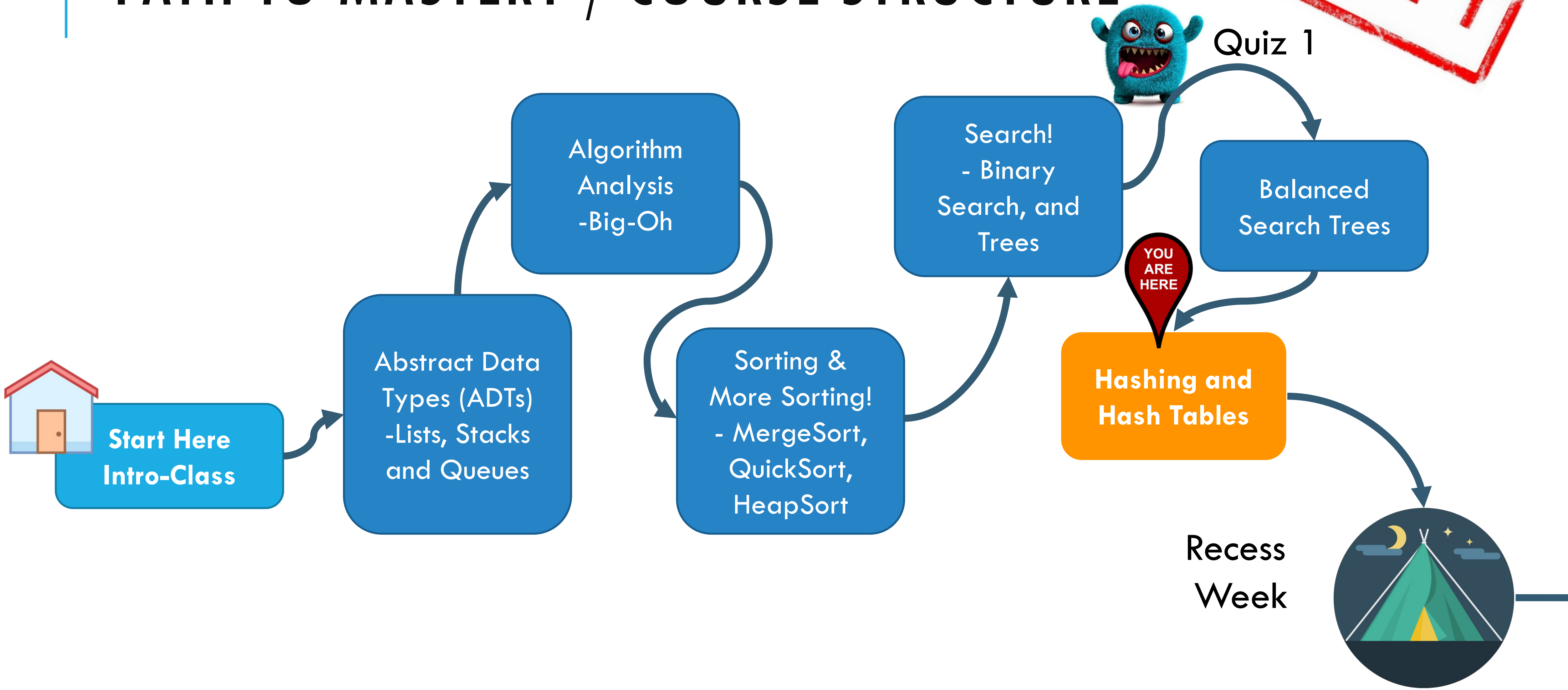
LECTURE 9: HASHING (PART 1)

Harold Soh
harold@comp.nus.edu.sg

PATH TO MASTERY / COURSE STRUCTURE

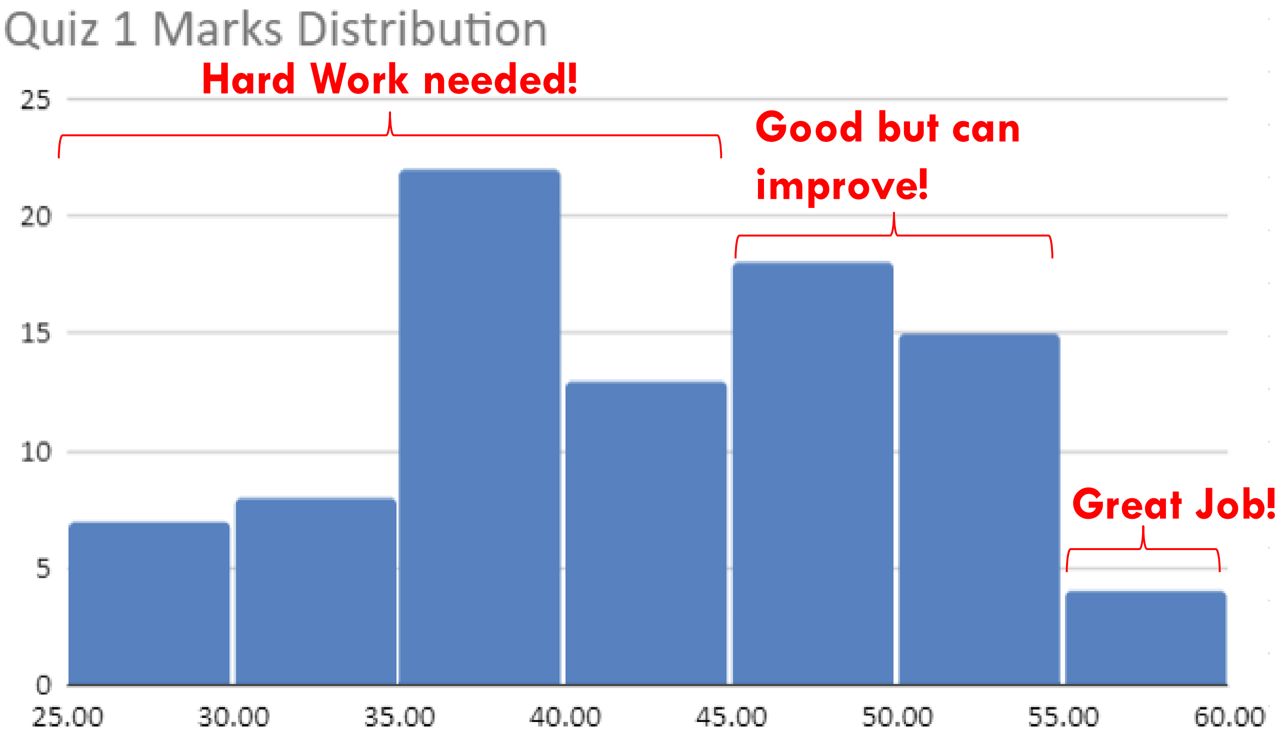
DRAFT



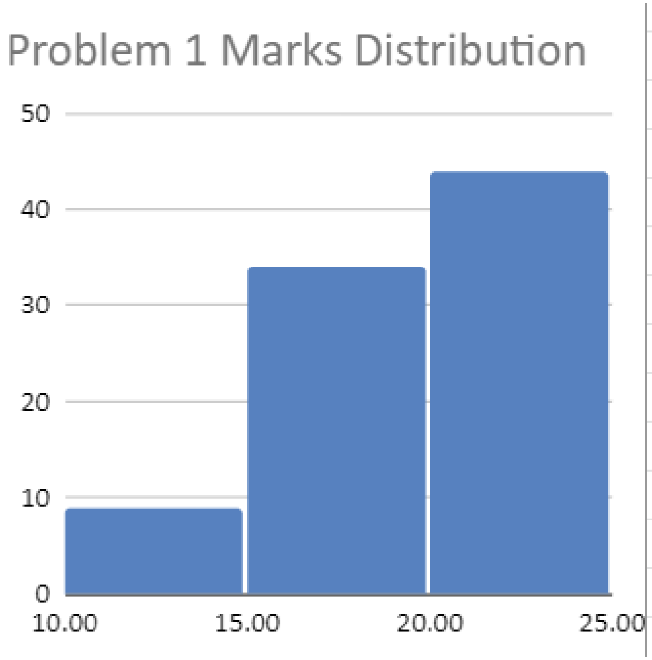
ADMINISTRATIVE ISSUES

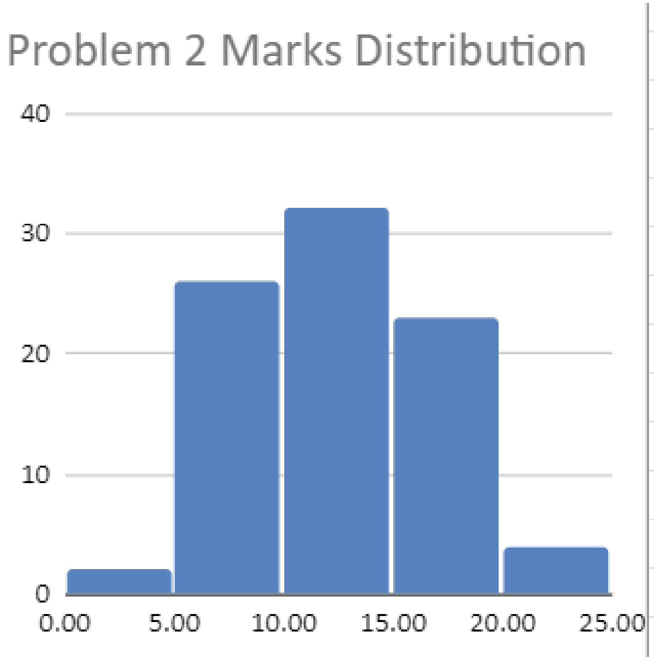
We have graded your Quiz 1.

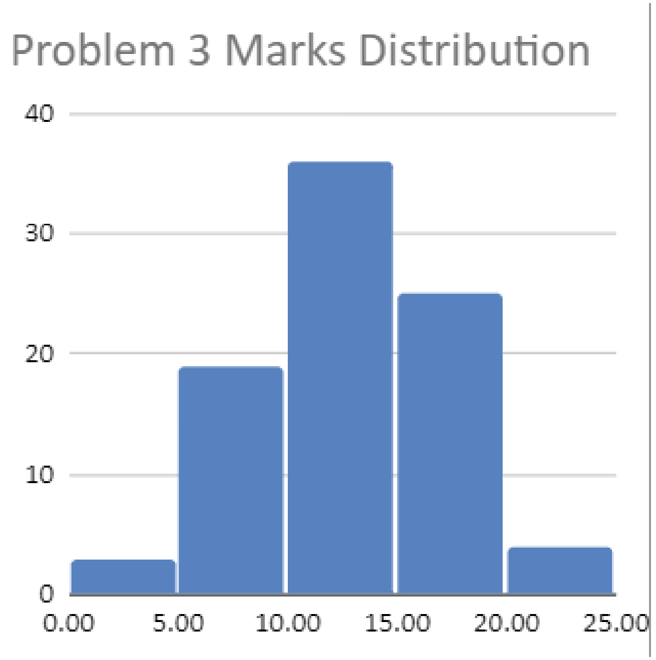
- Issues: See me after lecture today.
- Your grades will be up on Luminus by Friday.



DISTRIBUTION ACROSS PROBLEMS







REMEDIAL CLASSES

Remedial on Recurrence Relations

- Yesterday at Embedded Systems lab
- Today at SR5 (COM1-02-01)

Remedial on Quiz 2 Prep

- During recess week

ADMINISTRATIVE ISSUES

Quiz 2 is after recess week

- Tuesday (1st Oct) during Lecture.
- Open-book quiz
- No electronic equipment allowed.
- Will cover everything up to Balanced BSTs (BBSTs)
  - No hashing.

QUESTIONS?

LEARNING OUTCOMES

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

- Describe the **Symbol Table ADT**
- Explain the **Hash Table** Data Structure
- Analyze **the performance of the Hash Table**
- Describe the differences between the **Chaining** and **Open Addressing**

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?

Design an ADT for this problem

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?

Design an ADT for this problem

THE DICTIONARY ADT

Operations with k = key, v = value:

`insert(k, v)`: inserts an element with value v and key k

`search(k)`: returns the value with key k

`delete(k)`: deletes the element with key k

`contains(k)`: true if the dictionary contains an element with key k

`size()`: returns the size of the dictionary

*also called "Symbol Tables"

THE ORDERED DICTIONARY ADT

Operations with k = key, v = value:

`insert(k, v)`: inserts an element with value v and key k

`search(k)`: returns the value with key k

`delete(k)`: deletes the element with key k

`contains(k)`: true if the dictionary contains an element with key k

`floor(k)`: returns next key $\leq$ k

`ceiling(k)`: returns next key $\geq$ k

`size()`: returns the size of the dictionary

*also called "Ordered Symbol Tables"

THE ORDERED DICTIONARY ADT

Operations with k = key, v = value:

`insert(k, v)`: inserts an element with value v and key k

`search(k)`: returns the value with key k

`delete(k)`: deletes the element with key k

`contains(k)`: true if the dictionary contains an element with key k

`floor(k)`: returns next key $\leq$ k

`ceiling(k)`: returns next key $\geq$ k

`size()`: returns the size of the dictionary

*also called "Ordered Symbol Tables"

HOW CAN WE IMPLEMENT A SYMBOL TABLE?

Data Structure	Avg. Insert Time	Avg. Search Time
Unordered Array / Linked List	O(1)	O(n)
Ordered Array / Linked List	O(n)	O(log n)
Balanced Binary Search Tree (AVL)	O(log n)	O(log n)

HOW CAN WE IMPLEMENT A SYMBOL TABLE?

Data Structure	Avg. Insert Time	Avg. Search Time
Unordered Array / Linked List	O(1)	O(n)
Ordered Array / Linked List	O(n)	O(log n)
Balanced Binary Search Tree (AVL)	O(log n)	O(log n)
Hash Table	**O(1)**	**O(1)**

BUT O(1)? HOW IS THAT POSSIBLE?

Doesn't that mean we can sort in O(n)?

Fact: The fastest we can find something *using order comparisons* is $O(n \log n)$.

Conclusion: a hash table does not work using order comparisons!

WHAT DO WE GIVE UP?

Data Structure	Avg. Insert Time	Avg. Search Time	Avg. Max/Min	Avg. Floor/ Ceiling
Unordered Array / Linked List	O(1)	O(n)	O(n)	O(n)
Ordered Array / Linked List	O(n)	O(1)	O(1)	O(log n)
Balanced Binary Search Tree (AVL)	O(log n)	O(log n)	O(log n)	O(log n)
Hash Table	**O(1)**	**O(1)**		

WHAT DO WE GIVE UP?

no free lunch!

Data Structure	Avg. Insert Time	Avg. Search Time	Avg. Max/Min	Avg. Floor/ Ceiling
Unordered Array / Linked List	O(1)	O(n)	O(n)	O(n)
Ordered Array / Linked List	O(n)	O(1)	O(1)	O(log n)
Balanced Binary Search Tree (AVL)	O(log n)	O(log n)	O(log n)	O(log n)
Hash Table	**O(1)**	**O(1)**	**O(n)**	**O(n)**

LAST LECTURE'S PROBLEM: POVERTY IDENTIFICATION

The Stop-Poverty charity calls:

To provide financial aid, Help identify families:

- earning exactly $a amount
- earning less than $a amount
- earning more than $a amount

We can't do these efficiently with a standard hash table!

WHAT CAN HASH TABLES DO?

An Actual Dictionary (Spell checker/Autocorrect!)

Phone Book

Internet DNS

Singapore ID Database

Java Compiler

How can we implement a hash table?

A TRIVIAL HASH TABLE: DIRECT ACCESS TABLES

Just use a straightforward table.

Index objects by integer keys.

O(1) insert, O(1) search!

Awesome!

A TRIVIAL HASH TABLE: DIRECT ACCESS TABLES

Just use a straightforward table.

Index objects by integer keys.

O(1) insert, O(1) search!

~~**Awesome!**~~

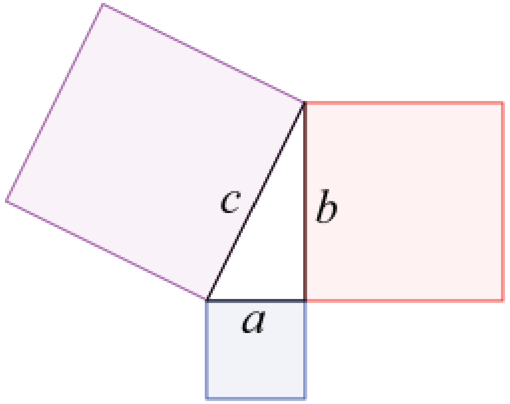
not really

Problems:

1. Too much space! If all integers are possible, what's the size? $2^{64} - 1$ **this is a heck a lot of memory!**
2. What if keys are not integers?
   - ("CS2040S", "is awesome!")
   - (3.14159, "I like pie")

THE SECOND PROBLEM IS EASIER:

Pythagoras said: “Everything is a number”

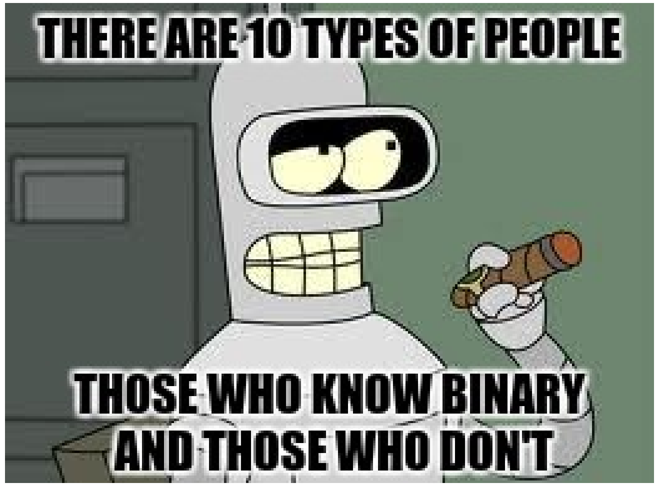


“The School of Athens” by Raphael

[Source: MIT 6.006]

https://en.wikipedia.org/wiki/The_School_of_Athens

THE SECOND PROBLEM IS EASIER:



Pythagoras said: "Everything is a number"

Especially true in a computer

Everything is a binary number: a sequence of bits!

English:

- 26 letters => 5 bits/letter
- Longest word = 28 letters ("antidisestablishmentarianism")
- 28 letters * 5 bits = 140 bits
- So we can store any possible English word in a direct-access array of size 2^{140}.

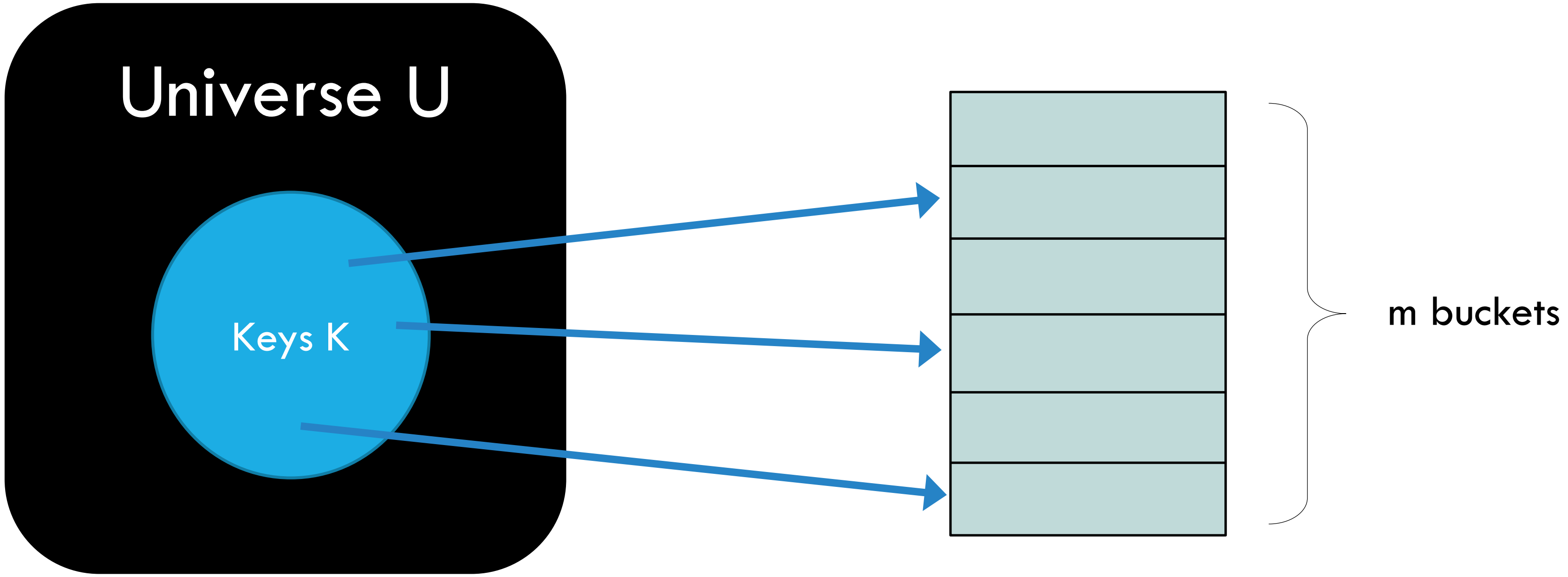
≈ number of atoms in planet Earth

HASH FUNCTIONS

Problem: U, the space of *possible* keys, is HUUUUGGGEEEE!

BUT: the space of *actual* keys is not so large (n).

Question: How to map n key to m buckets?

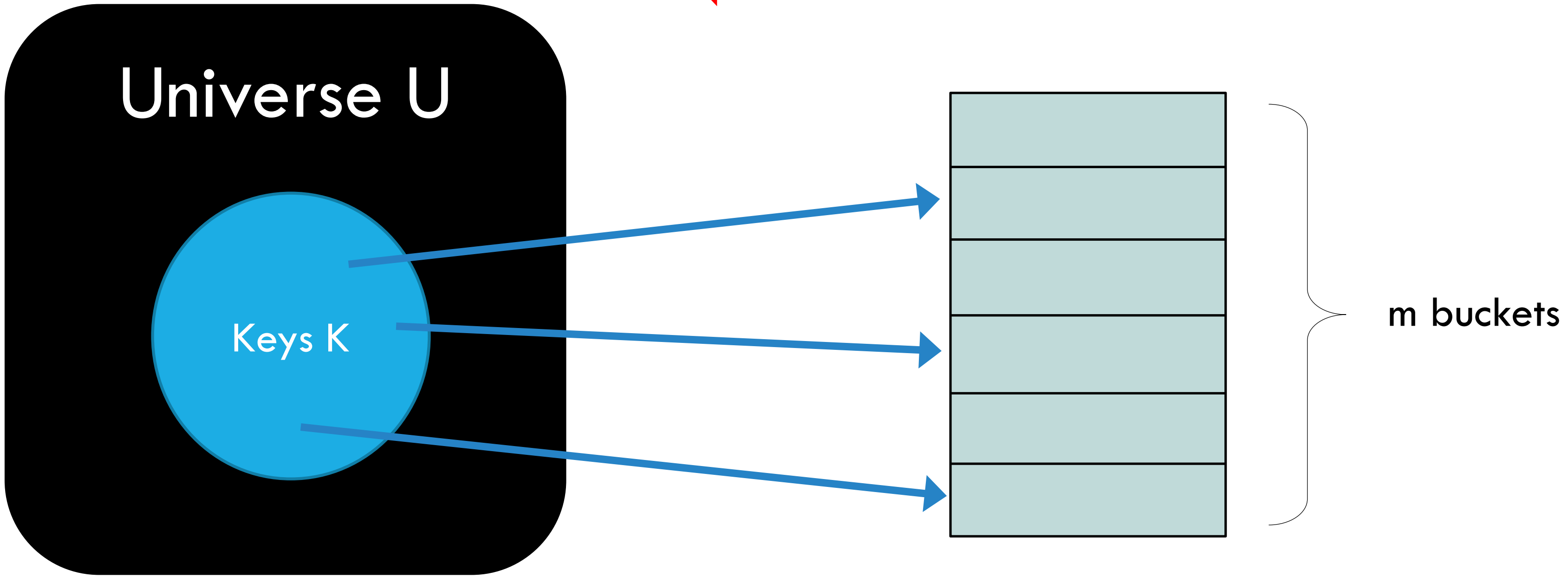


HASH FUNCTIONS

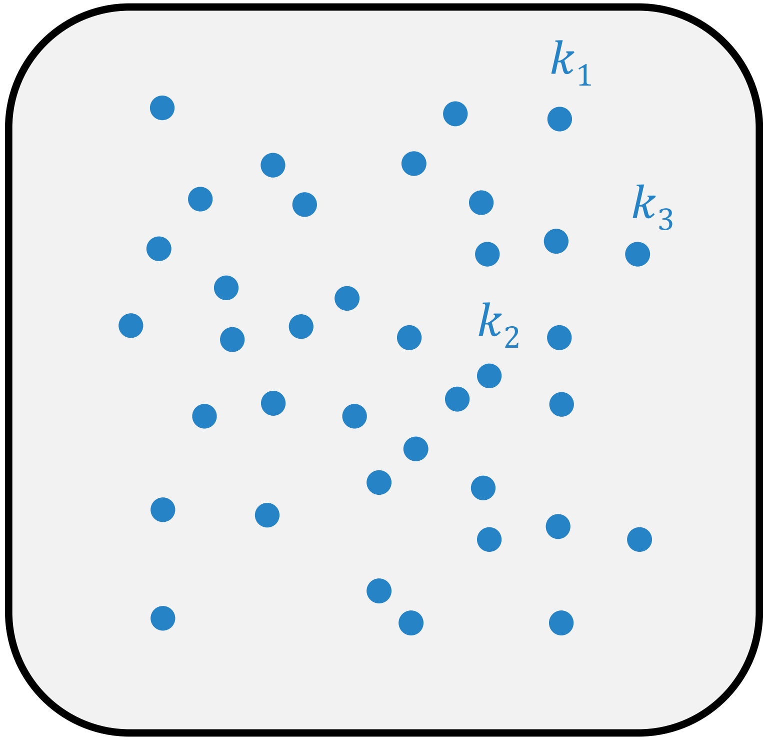
Define a hash function $h: U \to \{0, \ldots, m-1\}$

- Store key k in bucket $h(k)$
- Time complexity: Time to compute h + Time to access bucket
- Assume: computing h takes $O(1)$

This may not be true in practice!



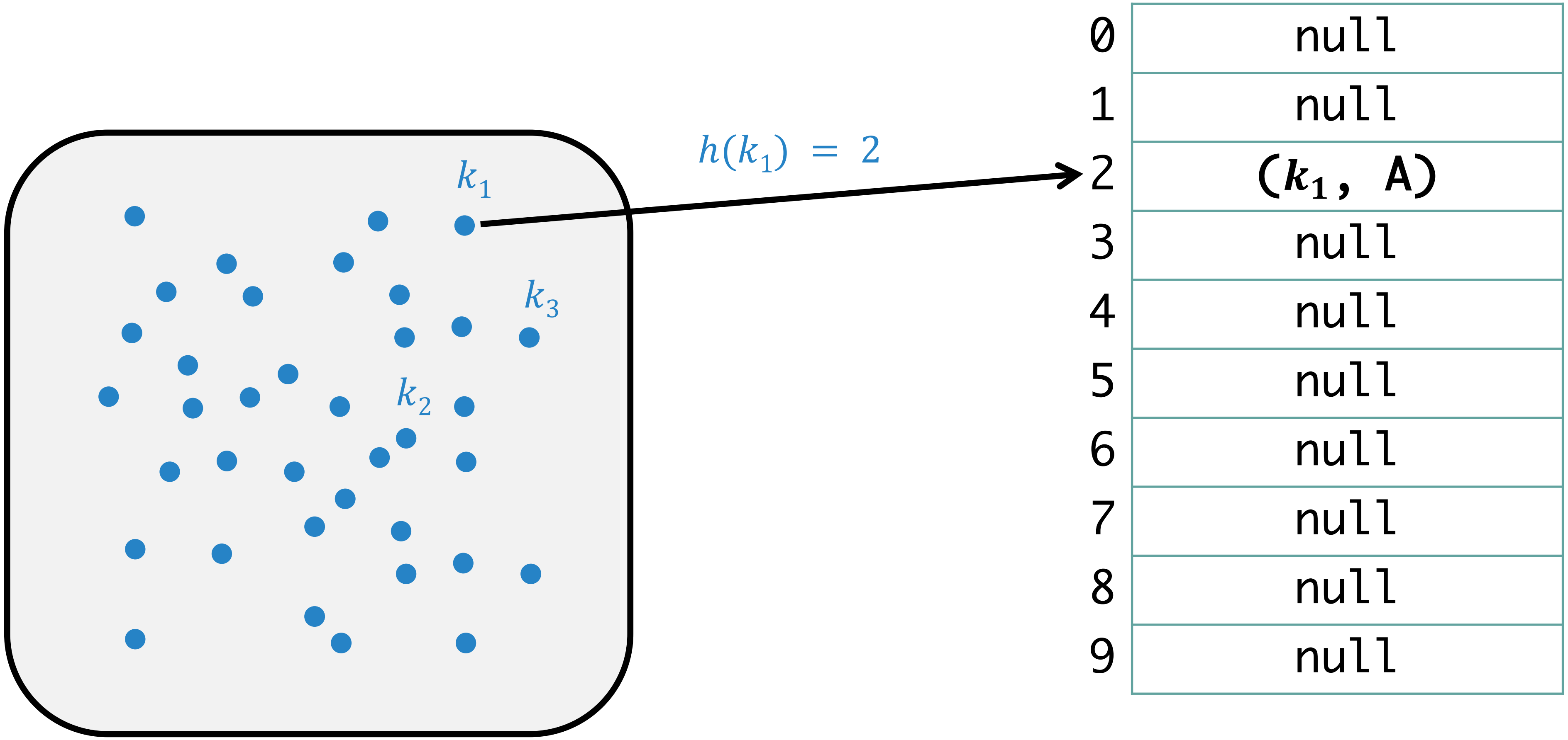
HASHING EXAMPLE



0	null
1	null
2	null
3	null
4	null
5	null
6	null
7	null
8	null
9	null

HASHING EXAMPLE

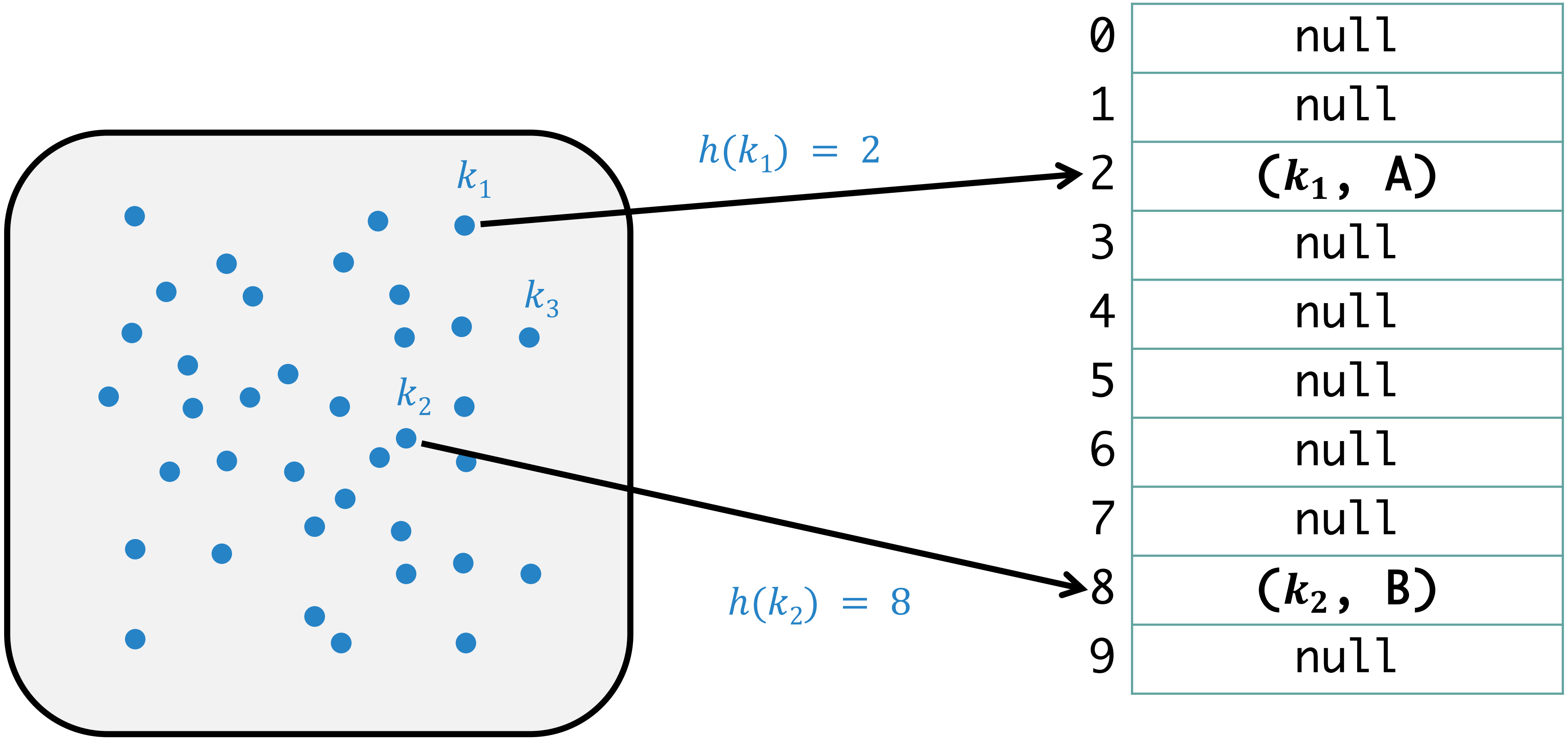
insert(k_1, A)



HASHING EXAMPLE

insert(k_1, A)

insert(k_2, B)



HASHING EXAMPLE

Collision!

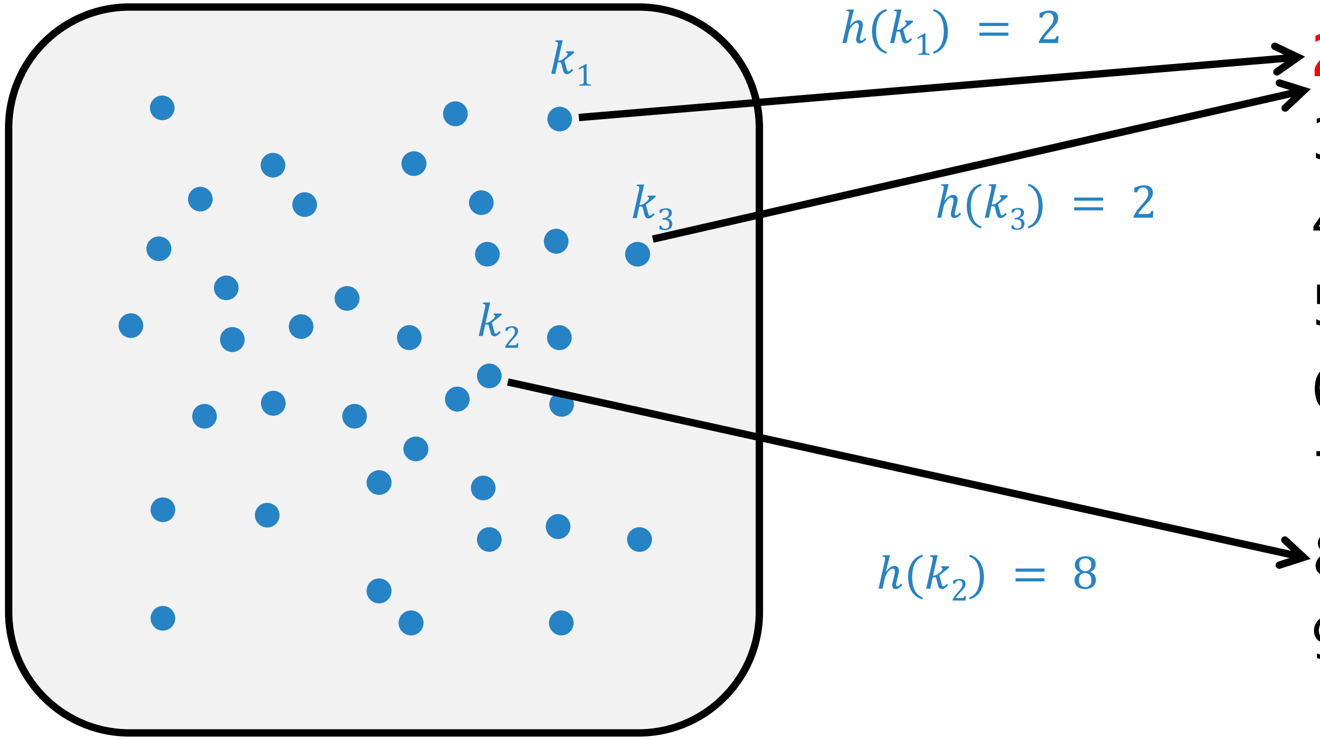
Two distinct keys k_1 and k_2 **collide** if:

$$h(k_1) = h(k_2)$$

insert(k_1, A)

insert(k_2, B)

insert(k_3, C)



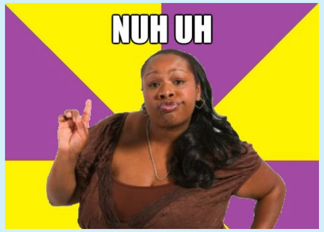
0	null
1	null
2	(k_1, A)
3	null
4	null
5	null
6	null
7	null
8	(k_2, B)
9	null

COLLISION FREE HASHING?

Assume a very large universe of possible keys (but we don't know the keys in advance).

Table size is smaller than the universe size (and number of actual keys).

Is it possible to derive a hashing function that has no collisions?

A. Yes! It's easy.
B. Sometimes, if we are really careful.
C. 

COLLISION FREE HASHING?

Assume a very large universe of possible keys (but we don't know the keys in advance).

Table size is smaller than the universe size (and number of actual keys).

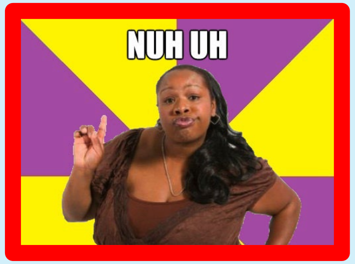
Pigeonhole principle: *If the number of buckets < number of keys, there must exist at least one bucket with more than 1 key.*

Is it possible to derive a hashing function that has no collisions?

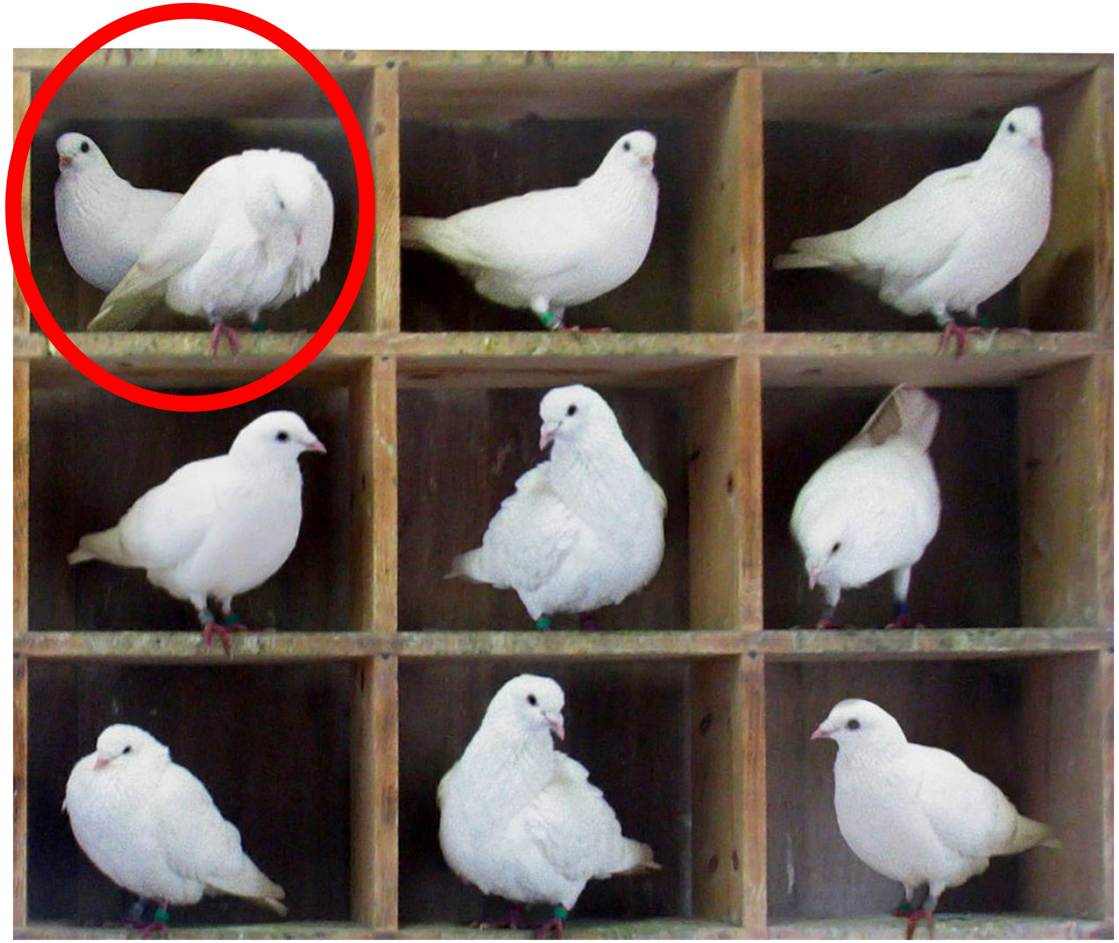
A. Yes! It's easy.

B. Sometimes, if we are really careful.

C.



10 PIGEONS IN 9 HOLES.



PUZZLE: HAIR ON THEIR HEADS?

Are there two people in Singapore with ***exactly*** the same number of hairs on their head (exclude bald people)?

Same number of Hairs?

A. Yes.
B. Maybe. Let me count.
C. No way!

PUZZLE: HAIR ON THEIR HEADS?

Are there two people in Singapore with ***exactly*** the same number of hairs on their head (exclude bald people)?

Yes! By the pigeonhole principle

There are 4.5 million people in Singapore.

But an average human only has 150,000 hairs!

Same number of Hairs?

A. **Yes.**
B. Maybe. Let me count.
C. No way!

LET'S ADD SOME ASSUMPTIONS!

Assume:

- we have n keys and $m \approx n$ buckets
- the keys are uniformly distributed.

What is the probability of collision in this case?

Poll Everywhere

https://bit.ly/2LvG9bq

PROBABILITY OF COLLISION

Consider we are hashing people by their birthdays (assume uniform).

m = 365 buckets (ignore year)

Question: How many people (keys) must be in a room (hash table) before the probability of two people sharing a birthday becomes at least 50%?

How many people in the room?

A. 12
B. 23
C. 111
D. 1100
E. Some even larger number!

PROBABILITY OF COLLISION

Poll Everywhere

https://bit.ly/2LvG9bq

Consider we are hashing people by their birthdays (assume uniform).

m = 365 buckets (ignore year)

Question: How many people (keys) must be in a room (hash table) before the probability of two people sharing a birthday becomes at least 50%?

Why?

How many people in the room?

A. 12
B. 23 (this is small!)
C. 111
D. 1100
E. Some even larger number!

PROBABILITY OF COLLISION

Poll Everywhere

https://bit.ly/2LvG9bq

Let $q(n)$ be the probability that everyone in the room has a *unique* birthday:

$$q(n) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{365-n+1}{365}$$

Then the probability of at least two people sharing the same birthday:

$$p(n) = 1 - q(n)$$

$$p(23) = 0.507 > 0.5$$

How many people in the room?

A. 12

B. 23 (this is small!)

C. 111

D. 1100

E. Some even larger number!

PROBABILITY OF COLLISION

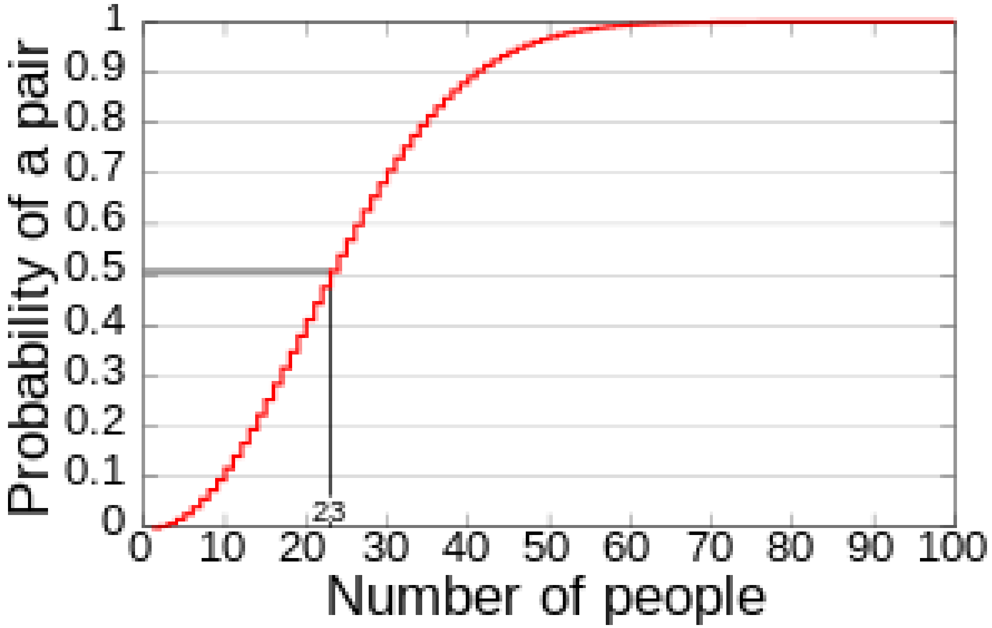
Let $q(n)$ be the probability that everyone in the room has a *unique* birthday:

$$q(n) = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{365-n+1}{365}$$

Then the probability of at least two people sharing the same birthday:

$$p(n) = 1 - q(n)$$

$$p(23) = 0.507 > 0.5$$



[from Wikipedia]

COLLISIONS **ARE A FACT OF LIFE**

If you don’t know the keys in advance.

- Otherwise, you can derive a perfect hash (google gperf)

Have a policy for handling collisions:

- Chaining (or Separate Chaining)
- Open Addressing

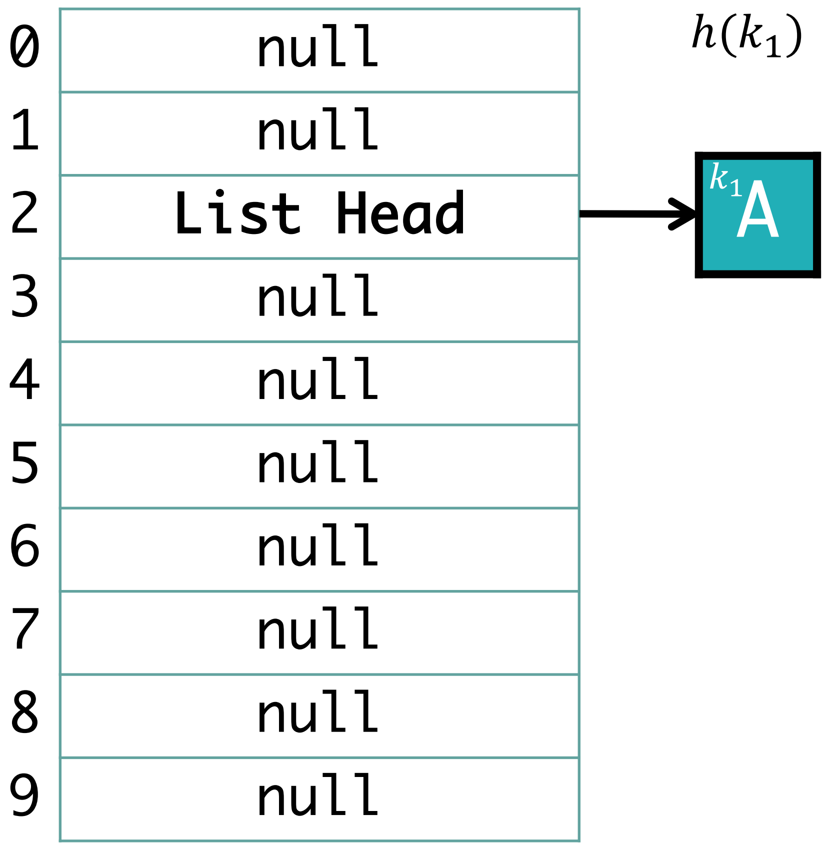
CHAINING

Idea: Each bucket stores a linked list.

If there is a collision, we add the item to the linked list.

insert(k_1, A)

$h(k_1) = 2$

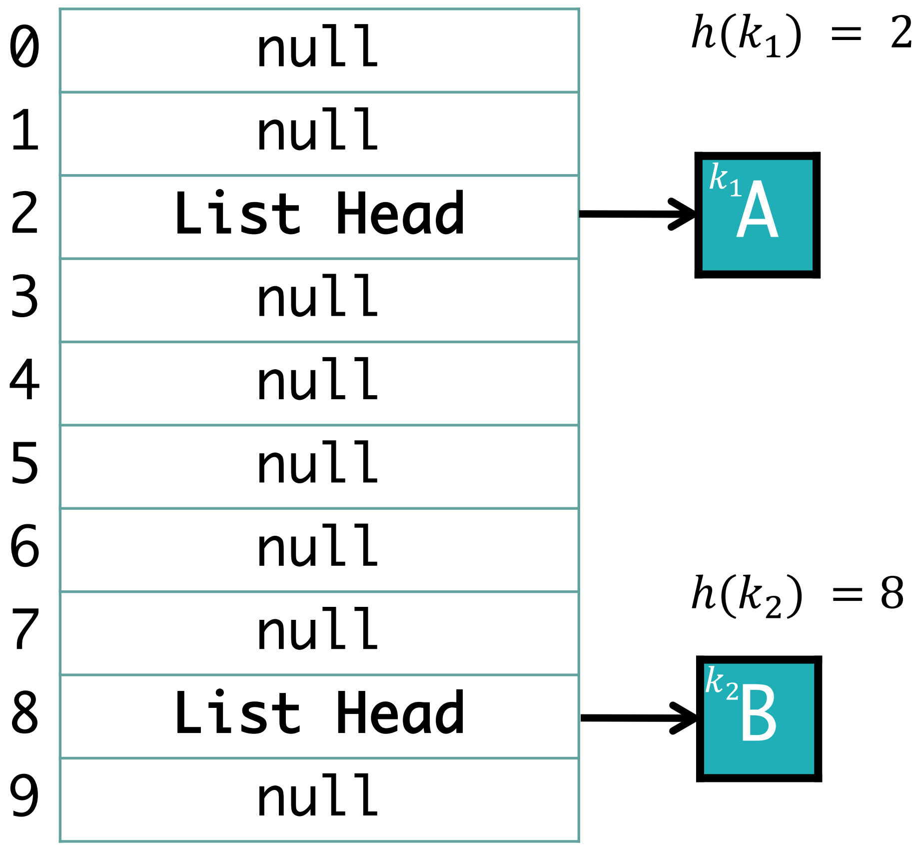


CHAINING

Idea: Each bucket stores a linked list.

If there is a collision, we add the item to the linked list.

insert(k_1, A)
insert(k_2, B)



CHAINING

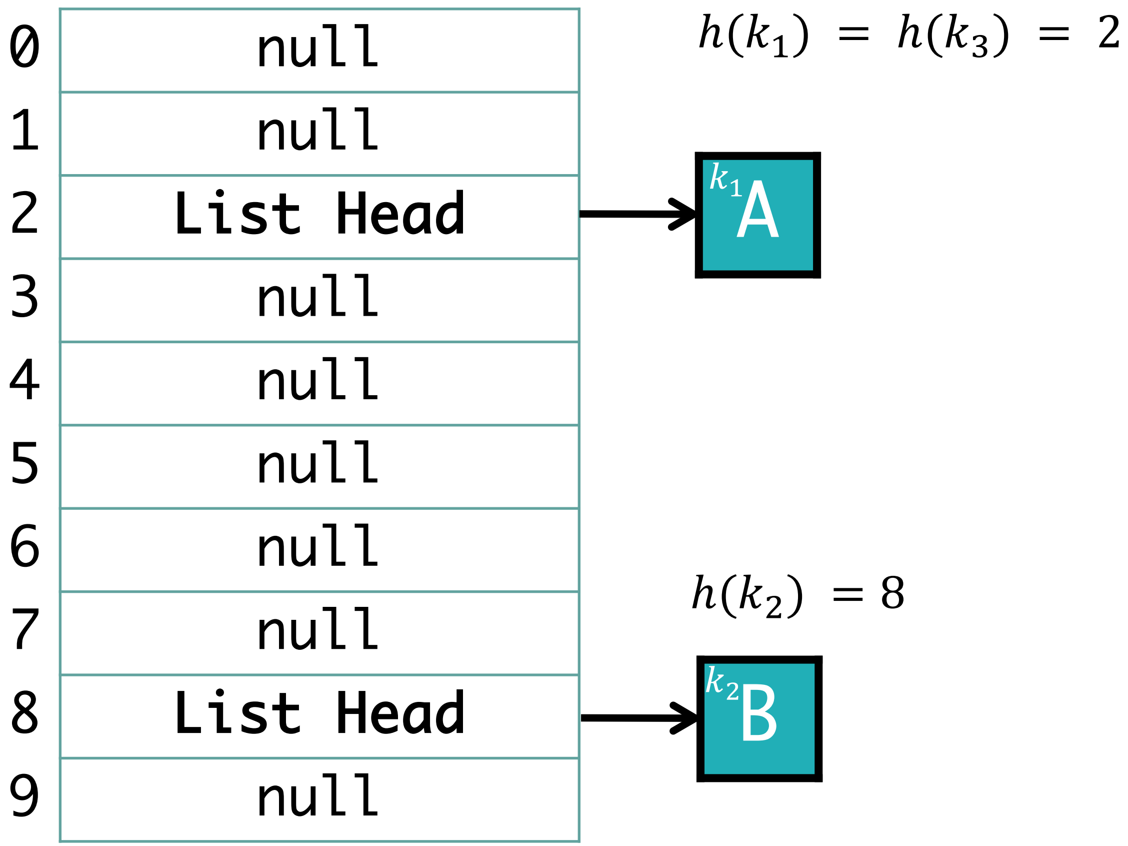
Collision.

but it's ok!

Idea: Each bucket stores a linked list.

If there is a collision, we add the item to the linked list.

insert(k_1, A)
insert(k_2, B)
insert(k_3, C)



CHAINING

Collision.

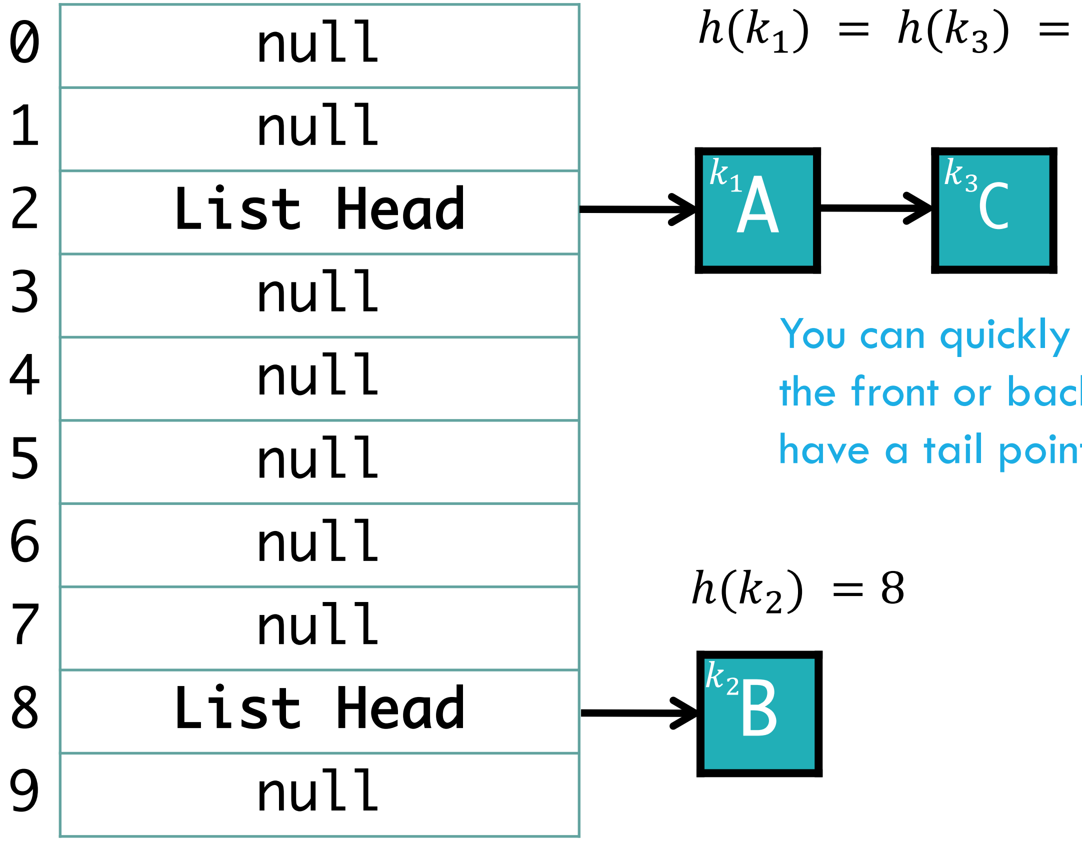
but it's ok!

Idea: Each bucket stores a linked list.

If there is a collision, we add the item to the linked list.

insert(k_1, A)
insert(k_2, B)
insert(k_3, C)

$h(k_1) = h(k_3) = 2$



You can quickly add k_3 to the front or back (if you have a tail pointer)

$h(k_2) = 8$

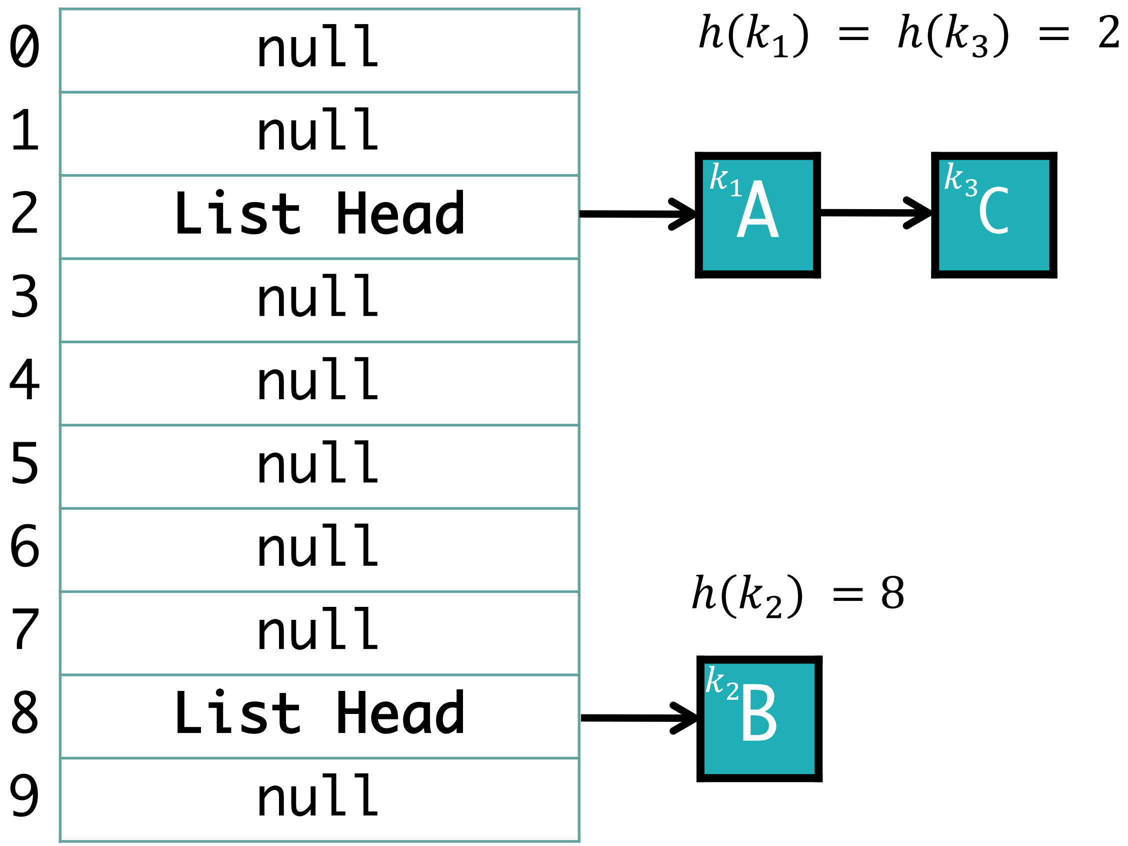
$$h(k_2) = 8$$

https://bit.ly/2LvG9bq

CHAINING: SPACE

What is the worst case space complexity for a hash table with separate chaining (m = table size, n = number of potential keys)?

A. $O(m)$
B. $O(n)$
C. $O(mn)$
D. $O(m + n)$
E. I know this one… I think… maybe…



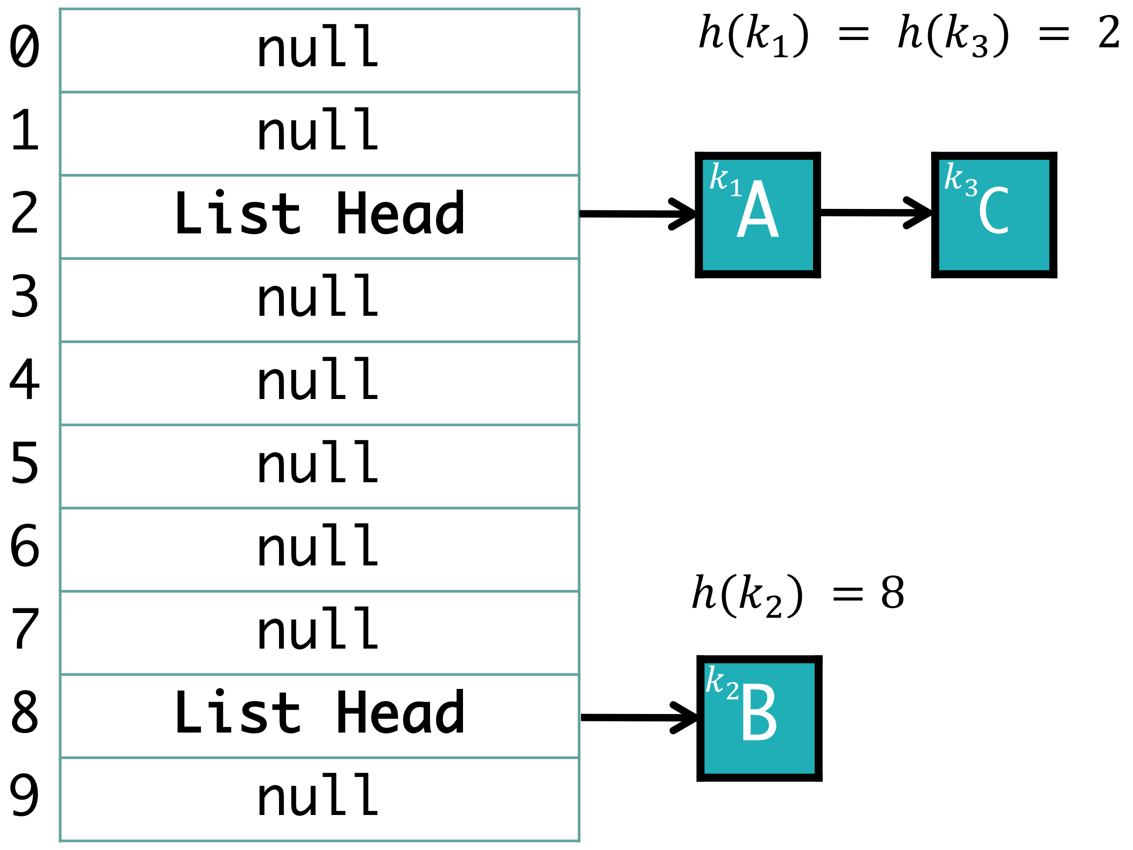
$$h(k_2) = 8$$

https://bit.ly/2LvG9bq

CHAINING: SPACE

What is the worst case space complexity for a hash table with separate chaining (m = table size, n = number of potential keys)?

A. $O(m)$
B. $O(n)$
C. $O(mn)$
D. $O(m + n)$
E. I know this one… I think… maybe…



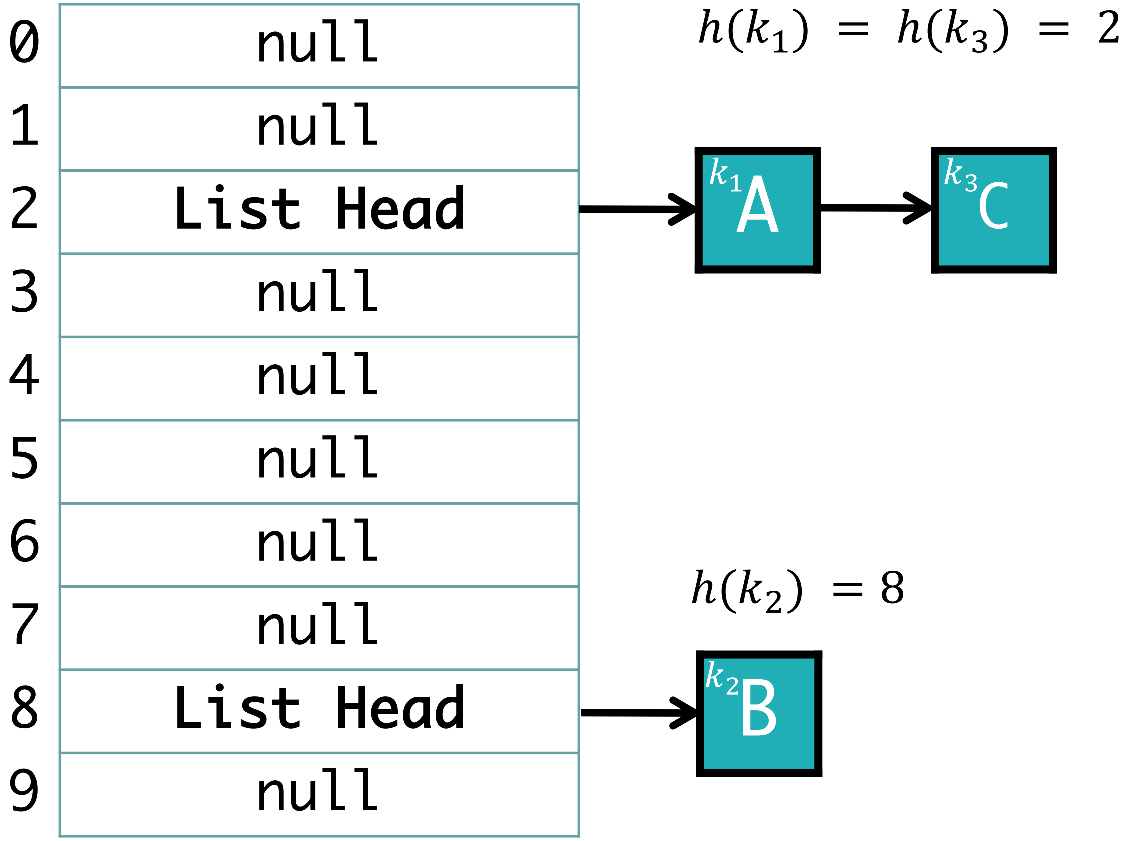
$$h(k_2) = 8$$

https://bit.ly/2LvG9bq

CHAINING: INSERT

How long does an insert take
m = table size, n = number of potential keys, Assume cost of $h = O(1)$. ?

A. $O(m)$
B. $O(n)$
C. $O(1)$
D. $O(m + n)$
E. I know this one… I think… maybe…



$h(k_2) = 8$

https://bit.ly/2LvG9bq

CHAINING: INSERT

How long does an insert take m = table size, n = number of potential keys, Assume cost of $h = O(1)$. ?

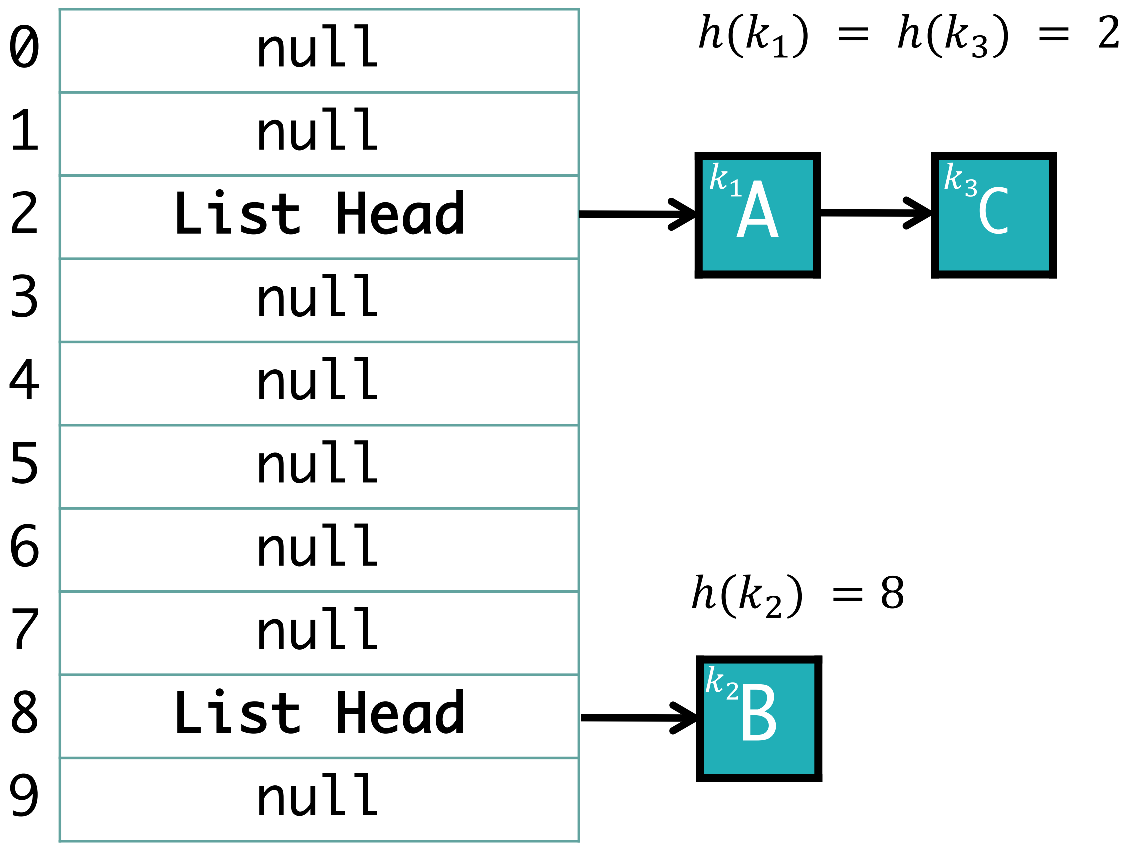
A. $O(m)$

B. $O(n)$

C. $O(1)$

D. $O(m + n)$

E. I know this one… I think… maybe…



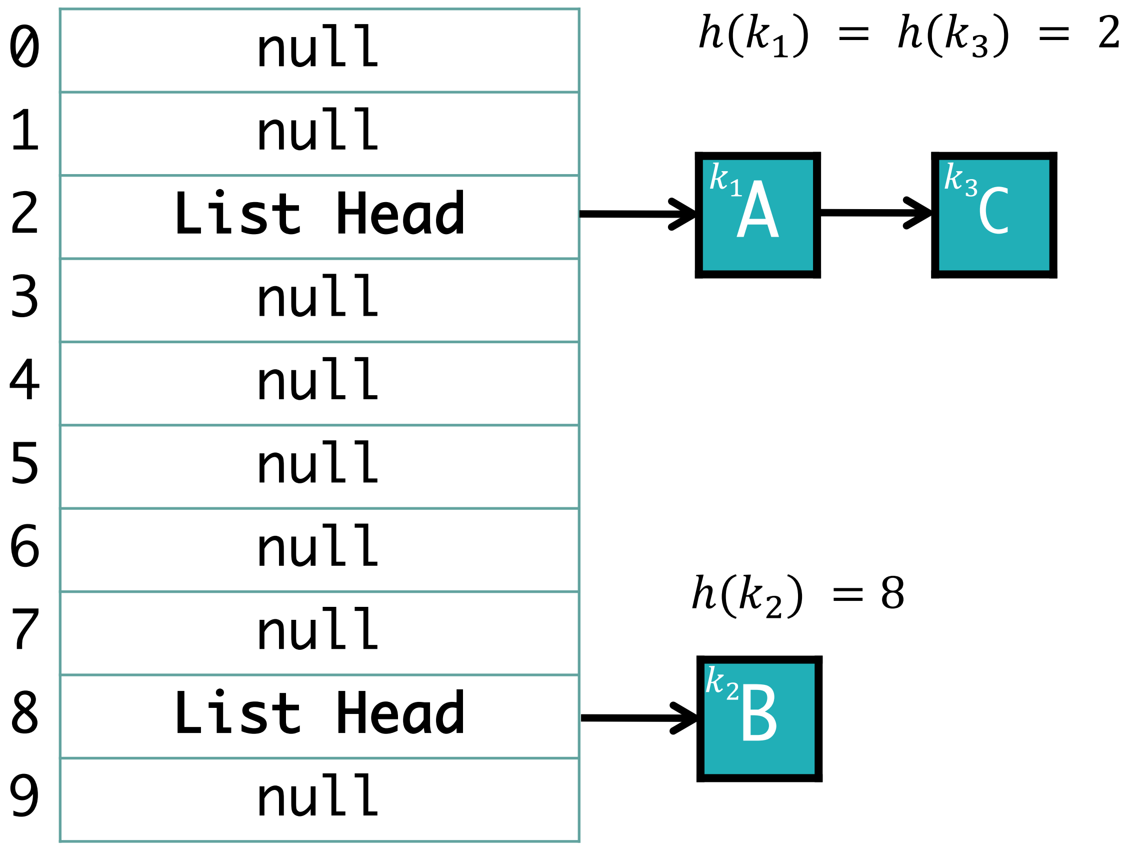
$h(k_2) = 8$

https://bit.ly/2LvG9bq

CHAINING: SEARCH

How long does an **search** take m = table size, n = number of potential keys, Assume cost of $h = O(1)$. ?

A. $O(m)$
B. $O(n)$
C. $O(1)$
D. $O(m + n)$
E. I know this one… I think… maybe…



Poll Everywhere

https://bit.ly/2LvG9bq

CHAINING: SEARCH

How long does an **search** take

m = table size, n = number of potential keys, Assume cost of $h = O(1)$. ?

A. $O(m)$

B. $O(n)$

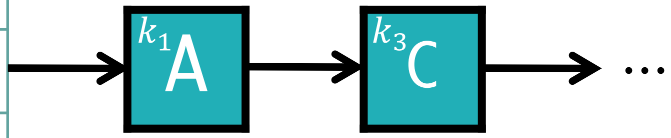
C. $O(1)$

D. $O(m + n)$

E. I know this one… I think… maybe…

0	null
1	null
2	List Head
3	null
4	null
5	null
6	null
7	null
8	null
9	null

$h(k_1) = h(k_2) = h(k_3) = \cdots$



all keys mapped to only 1 bucket!

SIMPLE UNIFORM HASHING ASSUMPTION

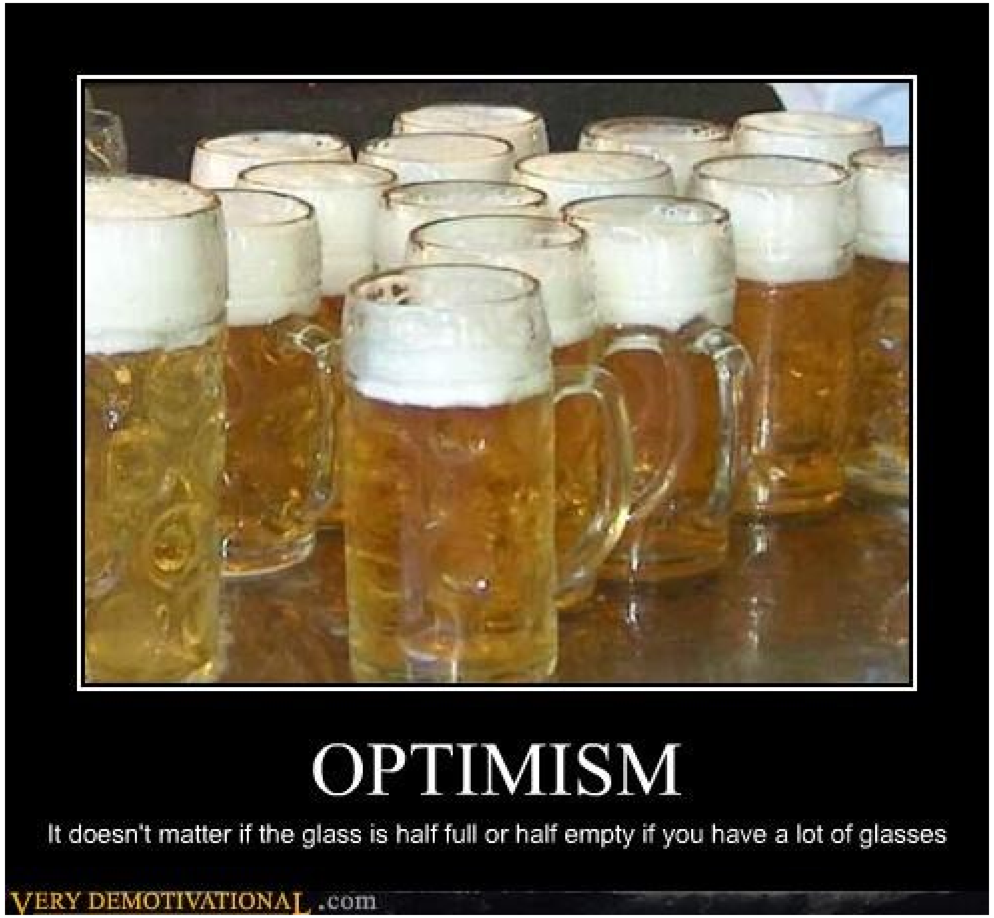
An **optimistic** assumption:

Every key is ***equally likely*** to map to every bucket

Keys are mapped ***independently***.

Intuition:

- Each key is put in a random bucket.
- As long as enough buckets, not too many keys in any one bucket.



Poll Everywhere

https://bit.ly/2LvG9bq

WHAT IS THE AVERAGE SEARCH TIME...

under the simple uniform hashing assumption (SUHA)

We have:

- m buckets
- n items
- Assume $n = \alpha m$ and $m \geq n$
- α is the "load factor"

$$\text{Expected search time} = \underbrace{1}_{\text{hashing + array access}} + \underbrace{\text{expected \# items per bucket}}_{\text{linked list traversal}}$$

What is the average search time under SUHA?

A. $O(m)$

B. $O(n)$

C. $O(1)$

D. $O(m + n)$

E. I thought we were doing some fingerprint stuff?

Poll Everywhere

https://bit.ly/2LvG9bq

WHAT IS THE AVERAGE SEARCH TIME…

under the simple uniform hashing assumption (SUHA)

We have:

- m buckets
- n items
- Assume $n = \alpha m$ and $m \geq n$
- α is the “load factor”

Expected search time $= \underbrace{1}_{\text{hashing + array access}} + \underbrace{\text{expected \# items per bucket}}_{\text{linked list traversal}}$

What is the average search time under SUHA?

A. $O(m)$

B. $O(n)$

C. $\boldsymbol{O(1)}$

D. $O(m + n)$

E. I thought we were doing some fingerprint stuff?

Why?

WHAT IS THE AVERAGE SEARCH TIME…

under the simple uniform hashing assumption (SUHA)

We have:

- m buckets
- n items
- Assume $n = \alpha m$ and $m \geq n$
- α is the “load factor”

Expected search time $= \underbrace{1}_{\text{hashing + array access}} + \underbrace{\text{expected \# items per bucket}}_{\text{linked list traversal}}$

Proof Sketch:

Indicator random variables

$X(i,j) = 1$ if item i is in bucket j

$X(i,j) = 0$ otherwise

Expected number of items in bucket b:

$$\mathbb{E}\left[\sum_{i}^{n} X(i,b)\right] = \sum_{i}^{n} \mathbb{E}[X(i,b)]$$

$$= \sum_i \frac{1}{m} = \frac{n}{m} = \alpha$$

Since $m > n$

$$\mathbb{E}\left[\sum_i X(i,b)\right] = O(1)$$

CHAINING

Insertion:

- Worst case: $O(1)$

Search:

- Expected Time: $O(1)$
- Worst-case Time: $O(n)$

Question: What is the expected maximum cost with n items?

Hint: Randomly throw n balls into $m = n$ bins. What is the maximum number of balls in a bin?

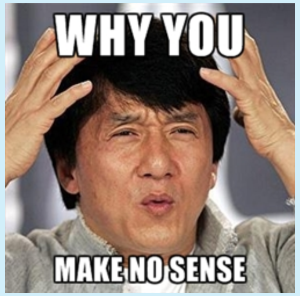
What is the *expected maximum* cost of a search?

A. $O(m)$

B. $O(n)$

C. $O(1)$

D. $O(\log n)$

E. 

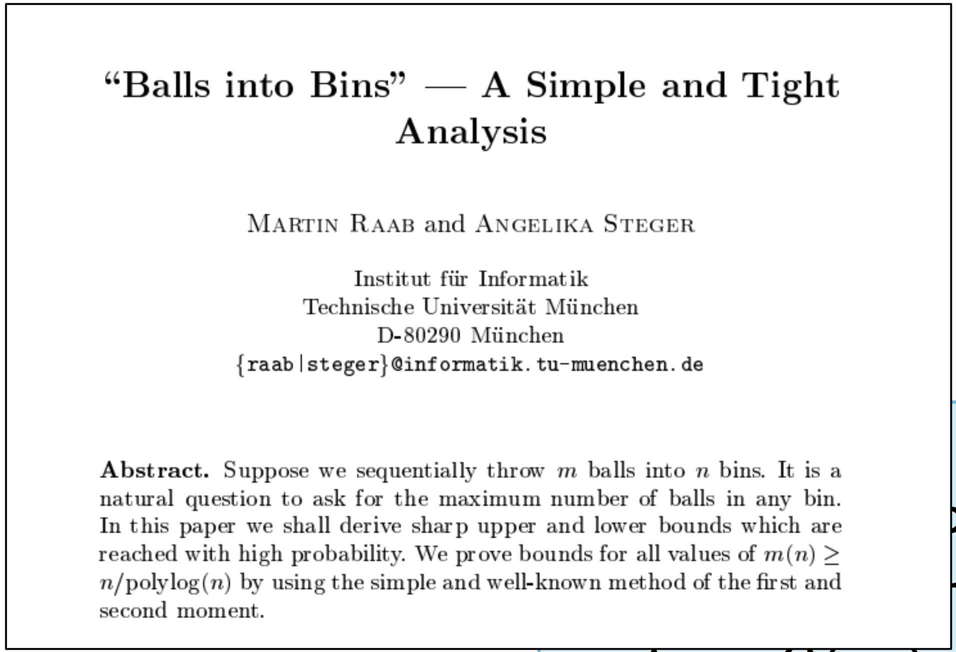
CHAINING

Insertion:
- Worst case: $O(1)$

Search:
- Expected Time: $O(1)$
- Worst-case Time: $O(n)$

Question: What is the expected maximum cost with n items?

Hint: Randomly throw n balls into $m = n$ bins. What is the maximum number of balls in a bin?



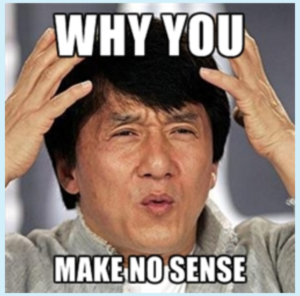
"Balls into Bins" — A Simple and Tight Analysis

Martin Raab and Angelika Steger

Institut für Informatik
Technische Universität München
D-80290 München
{raab|steger}@informatik.tu-muenchen.de

Abstract. Suppose we sequentially throw m balls into n bins. It is a natural question to ask for the maximum number of balls in any bin. In this paper we shall derive sharp upper and lower bounds which are reached with high probability. We prove bounds for all values of $m(n) \geq n/\text{polylog}(n)$ by using the simple and well-known method of the first and second moment.

...ected maximum
...n?

A. $O(m)$
B. $O(n)$
C. $O(1)$
D. $\boldsymbol{O(\log n)}$
E. 

CHAINING: OTHER AUXILIARY DATA STRUCTURES

Can use other data structures (what other data structures can you use?).

What are the pros and cons?

Improve worst case bounds. AVL is $O(\log n)$

BUT more overhead (memory + computation time) for a small number of items.

COLLISIONS ARE A FACT OF LIFE

If you don't know the keys in advance.

- Otherwise, you can derive a perfect hash (google gperf)

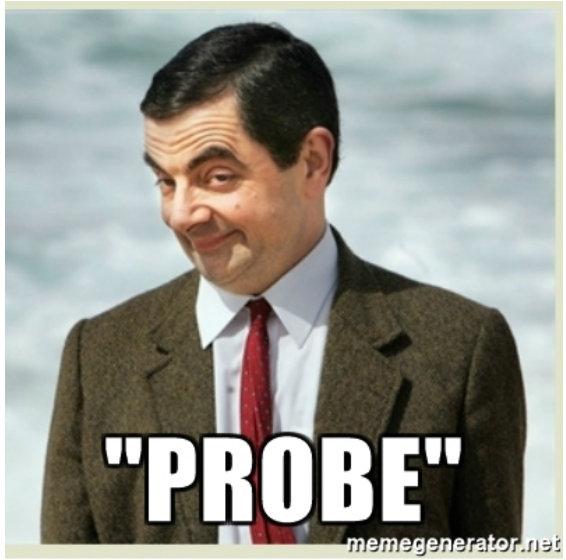
Have a policy for handling collisions:

- Chaining (or Separate Chaining)
- Open Addressing

OPEN ADDRESSING

Idea: On collision, probe until you find an empty slot.

Question: How to probe?



*Sorry, couldn’t resist some juvenile humor.

OPEN ADDRESSING

Idea: On collision, probe until you find an empty slot.

Question: How to probe?

`insert(`k_5`, E)`

$h(\boldsymbol{k_5}) = 2$

Collision!

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

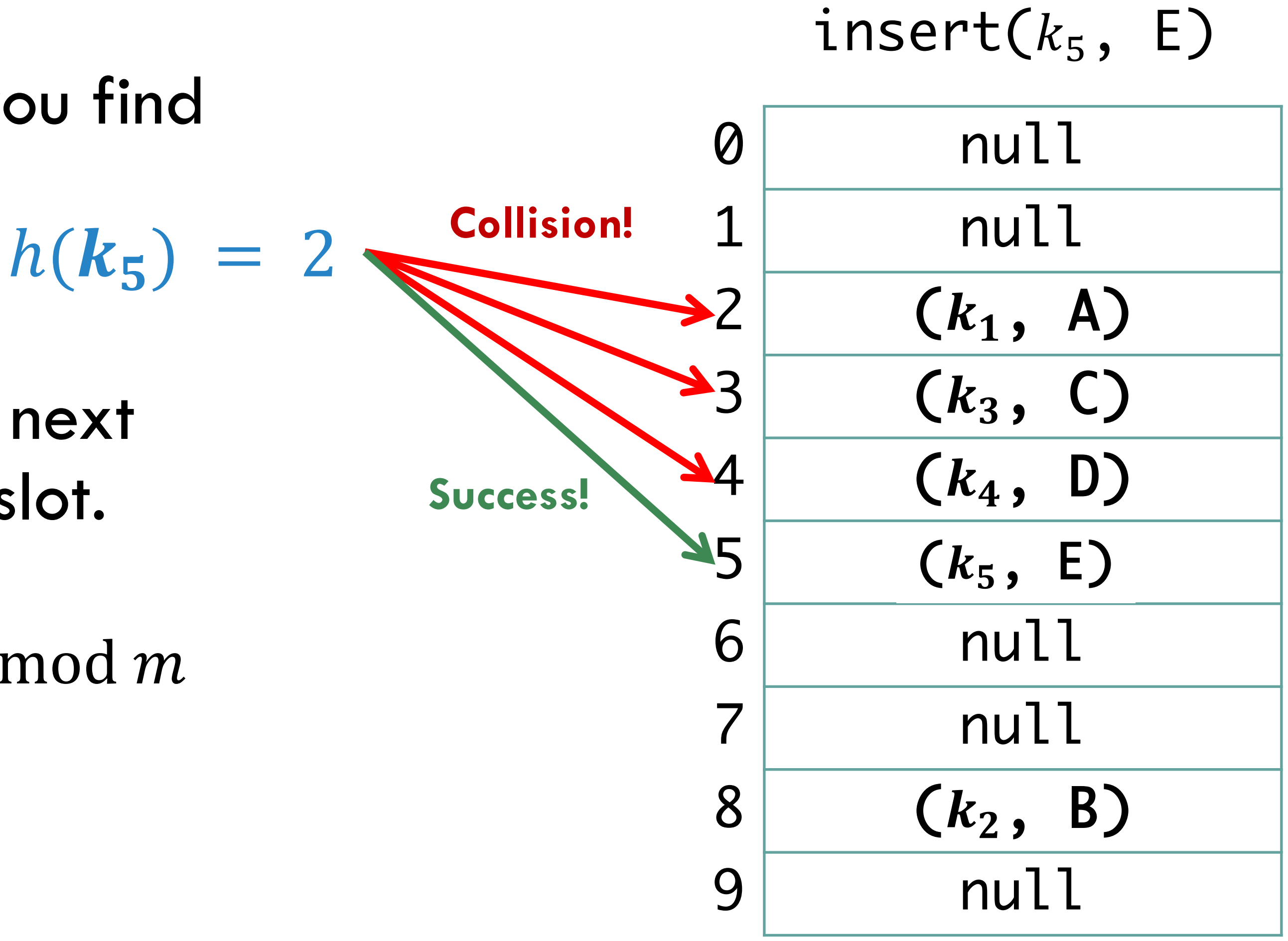
OPEN ADDRESSING: LINEAR PROBING

Idea: On collision, probe until you find an empty slot.

Question: How to probe?

Linear Probing: keep checking next bucket until you find an empty slot.

$$\text{index } i = \underbrace{(h(k)}_{\text{base address}} + \text{step} \times 1) \bmod m$$



OPEN ADDRESSING: LINEAR PROBING

Idea: On collision, probe until you find an empty slot.

Question: How to probe?

Linear Probing: keep checking next bucket until you find an empty slot.

index $i = (\underbrace{h(k)}_{\text{base address}} + \text{step} \times 1) \bmod m$

Modular Arithmetic

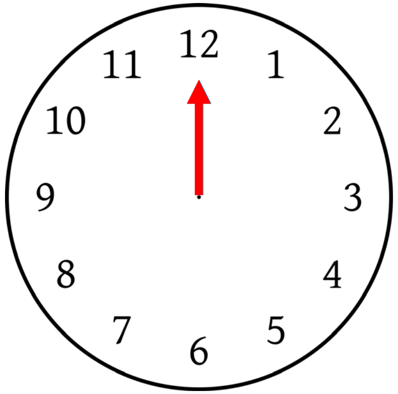
$\frac{a}{b} = q$ with remainder r

Then, using the **modulo operator**

$a \bmod b = r$

Example:

$48 \bmod 12 = 0$



OPEN ADDRESSING: LINEAR PROBING

Idea: On collision, probe until you find an empty slot.

Question: How to probe?

Linear Probing: keep checking next bucket until you find an empty slot.

$$\text{index } i = (\underbrace{h(k)}_{\text{base address}} + \text{step} \times 1) \bmod m$$

Modular Arithmetic

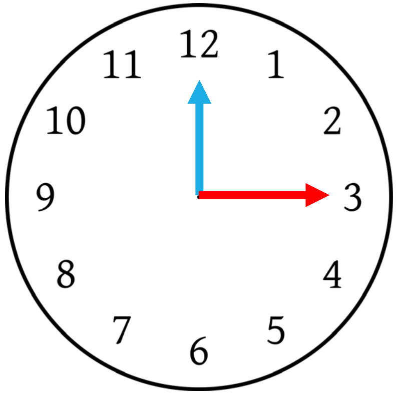
$\frac{a}{b} = q$ with remainder r

Then, using the **modulo operator**

$a \bmod b = r$

Example:

$$27 \bmod 12 = 3$$



OPEN ADDRESSING: LINEAR PROBING

Idea: On collision, probe until you find an empty slot.

Question: How to probe?

Linear Probing: keep checking next bucket until you find an empty slot.

index $i = \underbrace{(h(k)}_{\text{base address}} + \text{step} \times 1) \bmod m$

Modular Arithmetic

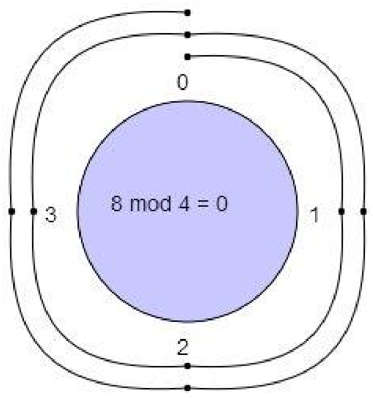
$\frac{a}{b} = q$ with remainder r

Then, using the **modulo operator**

$a \bmod b \ = \ r$

Example:

$$8 \bmod 4 = 0$$



OPEN ADDRESSING: LINEAR PROBING

Idea: On collision, probe until you find an empty slot.

Question: How to probe?

Linear Probing: keep checking next bucket until you find an empty slot.

$$\text{index } i = (\underbrace{h(k)}_{\text{base address}} + \text{step} \times 1) \bmod m$$

Modular Arithmetic

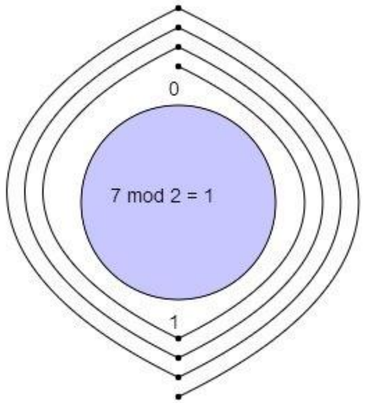
$\frac{a}{b} = q$ with remainder r

Then, using the **modulo operator**

$a \bmod b = r$

Example:

$$7 \bmod 2 = 1$$



OPEN ADDRESSING: LINEAR PROBING

Idea: On collision, probe until you find an empty slot.

Question: How to probe?

Linear Probing: keep checking next bucket until you find an empty slot.

$$\text{index } i = (h(k) + \text{step} \times 1) \bmod m$$

`insert(`k_5`, E)`

$h(\boldsymbol{k_5}) = 2$

Collision!

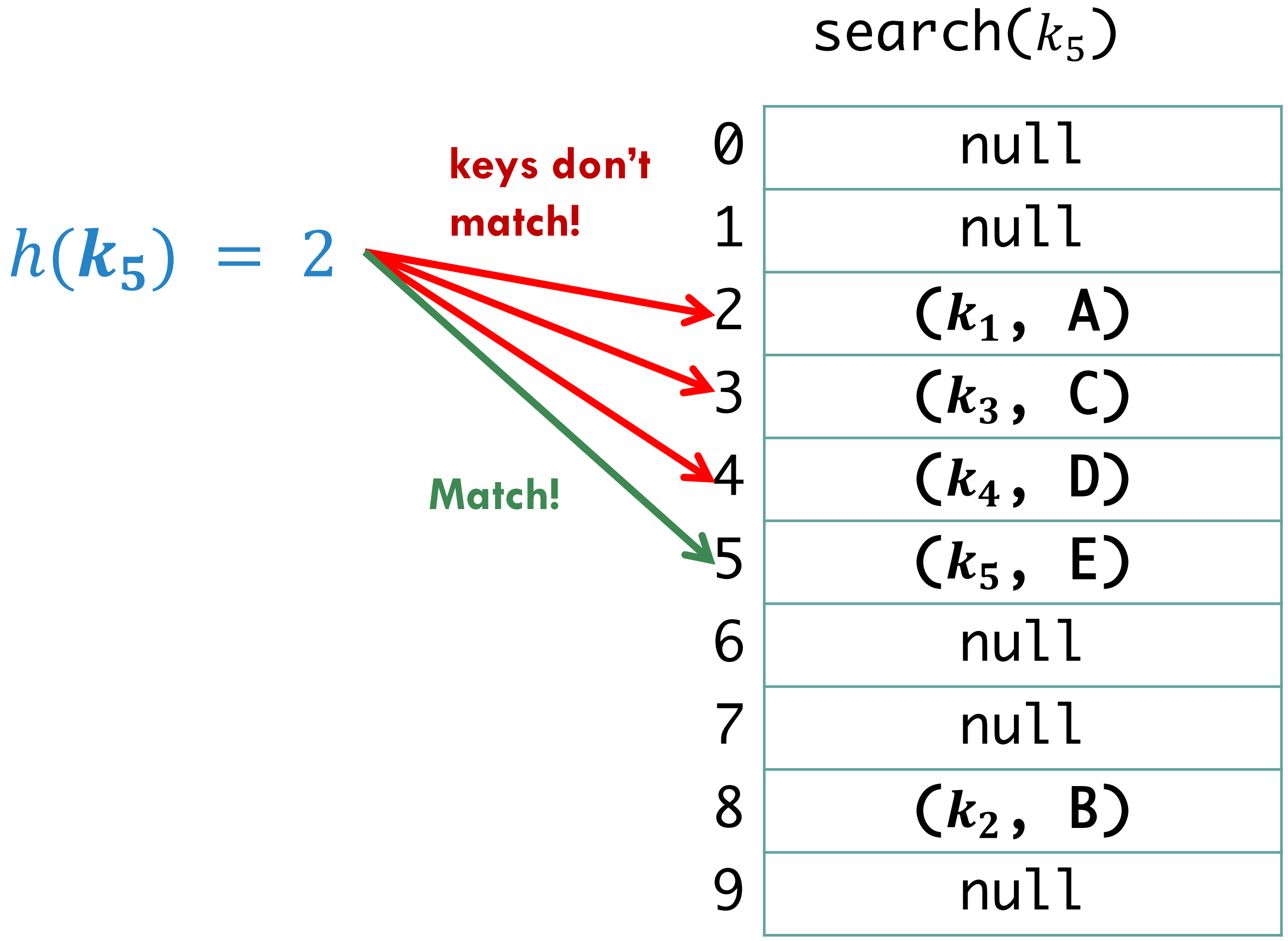
Success!

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

LINEAR PROBING: SEARCH

Search is similar to insert.

Keep probing until we find a key match.

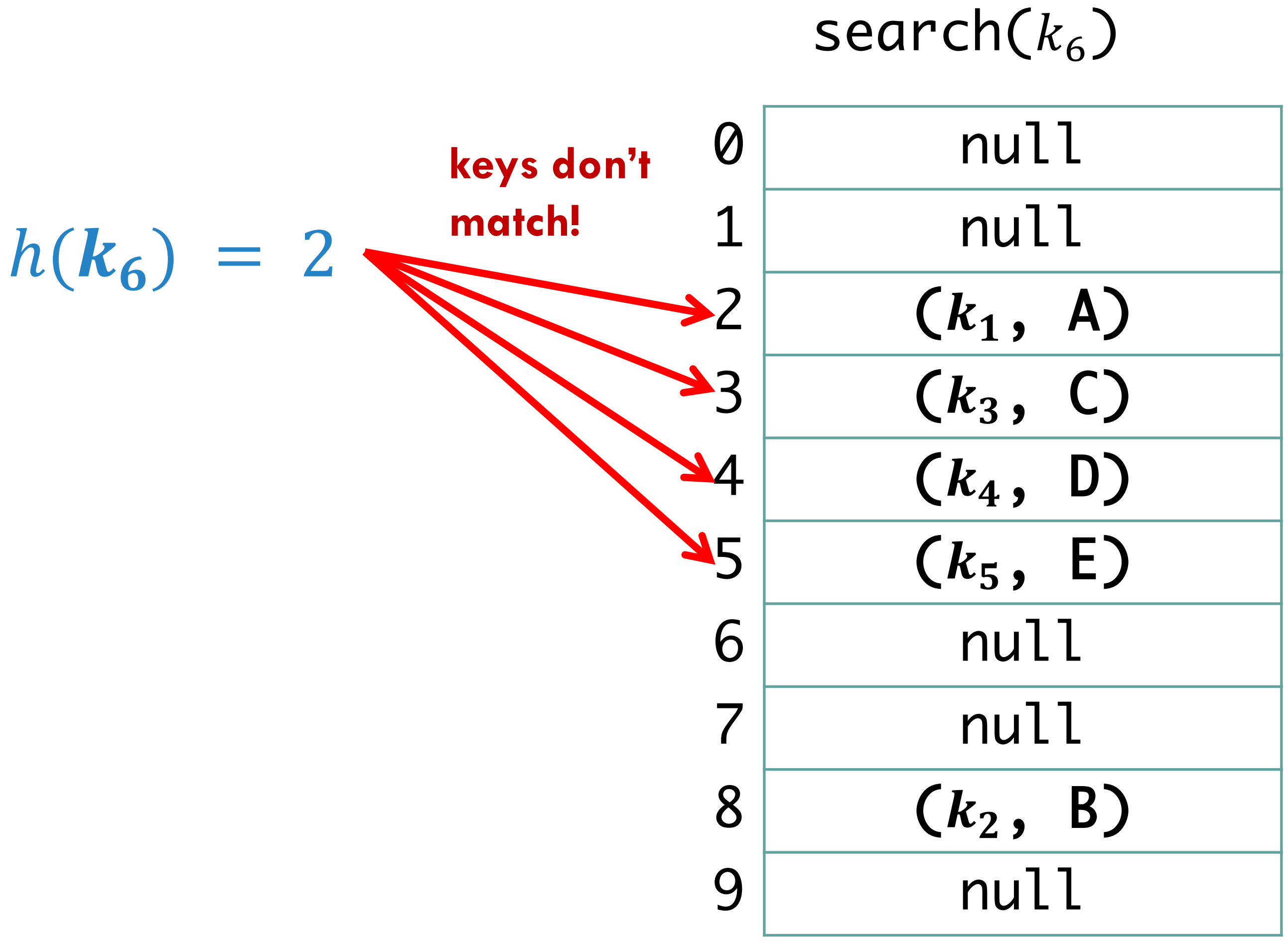


LINEAR PROBING: SEARCH

Search is similar to insert.

Keep probing until we find a key match.

When to stop?



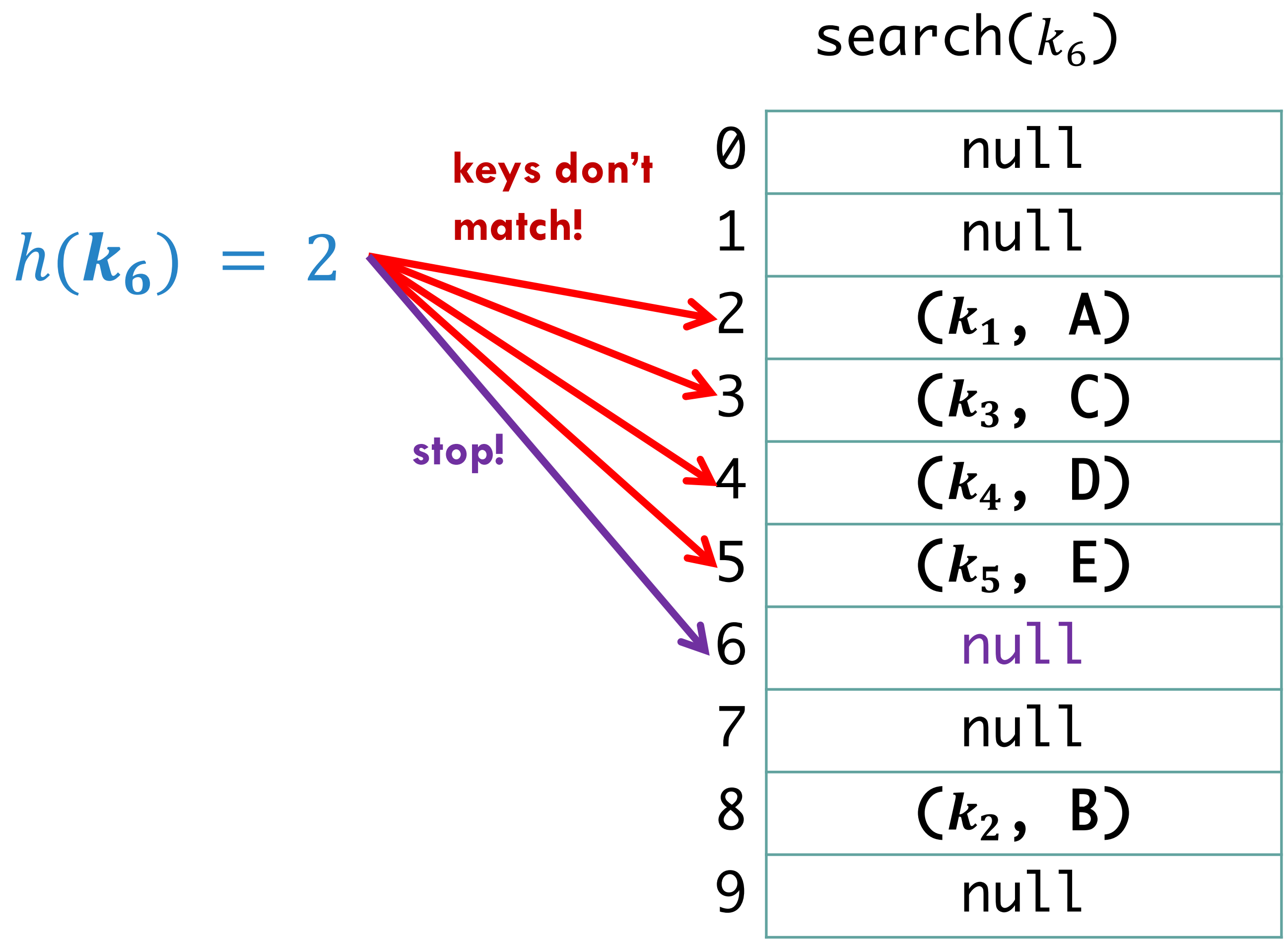
LINEAR PROBING: **SEARCH**

Search is similar to insert.

Keep probing until we find a key match.

When to stop?

When we hit a `null`



LINEAR PROBING: **DELETE**

Delete is easy.

Just do a search until you find the element and delete.

$h(\boldsymbol{k_5}) = 2$

keys don't match!

Match!

`delete(`k_5`)`

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

LINEAR PROBING: **DELETE**

Delete is easy.

Just do a search until you find the element and delete.

$h(\boldsymbol{k_5}) = 2$

keys don't match!

Match!

This is wrong! Why?

`delete(`k_5`)`

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

LINEAR PROBING: **DELETE**

Delete is easy.

Just do a search until you find the element and delete.

Consider:

$\texttt{delete}(k_3)$

$\texttt{search}(k_5)$

$h(\boldsymbol{k_3}) = 3$

$\texttt{delete}(k_3)$

0	null
1	null
2	$(\boldsymbol{k_1}, \text{A})$
3	$(\boldsymbol{k_3}, \text{C})$
4	$(\boldsymbol{k_4}, \text{D})$
5	$(\boldsymbol{k_5}, \text{E})$
6	null
7	null
8	$(\boldsymbol{k_2}, \text{B})$
9	null

LINEAR PROBING: **DELETE**

Delete is easy.

Just do a search until you find the element and delete.

Consider:

delete(k_3)

search(k_5)

$h(\boldsymbol{k_3}) = 3$

$h(\boldsymbol{k_5}) = 2$

keys don't match!

stop!

k_5 was not found even though it's in the hash table!

search(k_5)

0	null
1	null
2	(k_1, A)
3	null
4	(k_4, D)
5	$\boxed{(k_5, \text{E})}$
6	null
7	null
8	(k_2, B)
9	null

LINEAR PROBING: DELETE

Delete is easy.

Just do a search until you find the element and delete.

Delete uses a special symbol DEL.

search(k) continues even when seeing a DEL.

delete(k_3)

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

LINEAR PROBING: DELETE

Delete is easy.

Just do a search until you find the element and delete using the DEL symbol.

Consider:

$\texttt{delete}(k_3)$

$\texttt{search}(k_5)$

$\texttt{delete}(k_3)$

$$h(\boldsymbol{k_3}) = 3$$

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

LINEAR PROBING: **DELETE**

Delete is easy.

Just do a search until you find the element and delete using the DEL symbol

Consider:

delete(k_3)

search(k_5)

$h(\boldsymbol{k_3}) = 3$

$h(\boldsymbol{k_5}) = 2$

keys don't match!

Match!

search(k_5)

0	null
1	null
2	($\boldsymbol{k_1}$, A)
3	DEL
4	($\boldsymbol{k_4}$, D)
5	($\boldsymbol{k_5}$, E)
6	null
7	null
8	($\boldsymbol{k_2}$, B)
9	null

All ok now!

But what about inserts?

Just overwrite DEL

LINEAR PROBING: DELETE

Delete is easy.

Just do a search until you find the element and delete using the DEL symbol

Consider:

`delete(`k_3`)`

`search(`k_5`)`

`insert(`k_{42}`)`

$h(\boldsymbol{k_{42}}) = 3$

All ok now!

But what about inserts?

Just overwrite DEL

`insert(`k_{42}`, G)`

0	null
1	null
2	(k_1, A)
3	(k_{42}, G)
4	(k_4, D)
5	(k_5, E)
6	null
7	null
8	(k_2, B)
9	null

A PROBLEM: **CLUSTERS**

cluster = collection of consecutive occupied slots

With linear probing, are big or small clusters more likely to form?

A. Big
B. Small
C. Does the size actually matter?

0	null
1	A
2	A
3	A
4	A
5	A
6	null
7	null
8	B
9	B

A PROBLEM: **CLUSTERS**

cluster = collection of consecutive occupied slots

With linear probing, are big or small clusters more likely to form?

- **A. Big**
- B. Small
- C. Does the size actually matter?

0	null
1	A
2	
3	
4	
5	
6	null
7	null
8	B
9	

A PROBLEM: PRIMARY CLUSTERS

cluster = collection of consecutive occupied slots

In a hash table of size 10, consider 2 clusters:

- A: size 5
- B: size 2

Probability that a new inserted key k has a bucket in

- cluster A? 5/10
- cluster B? 2/10

<table>
<tr><td>0</td><td>null</td></tr>
<tr><td>1</td><td rowspan="5">A</td></tr>
<tr><td>2</td></tr>
<tr><td>3</td></tr>
<tr><td>4</td></tr>
<tr><td>5</td></tr>
<tr><td>6</td><td>null</td></tr>
<tr><td>7</td><td>null</td></tr>
<tr><td>8</td><td rowspan="2">B</td></tr>
<tr><td>9</td></tr>
</table>

A PROBLEM: PRIMARY CLUSTERS

cluster = collection of consecutive occupied slots

In a hash table of size 10, consider 2 clusters:

- A: size 5
- B: size 2

Probability that a new inserted key k has a bucket in

- cluster A? 5/10
- cluster B? 2/10

What happens after k is added?

Cluster grows by 1 element

0	null
1	A
2	A
3	A
4	A
5	A
6	null
7	null
8	B
9	B

A PROBLEM: PRIMARY CLUSTERS

cluster = collection of consecutive occupied slots

In a hash table of size 10, consider 2 clusters:

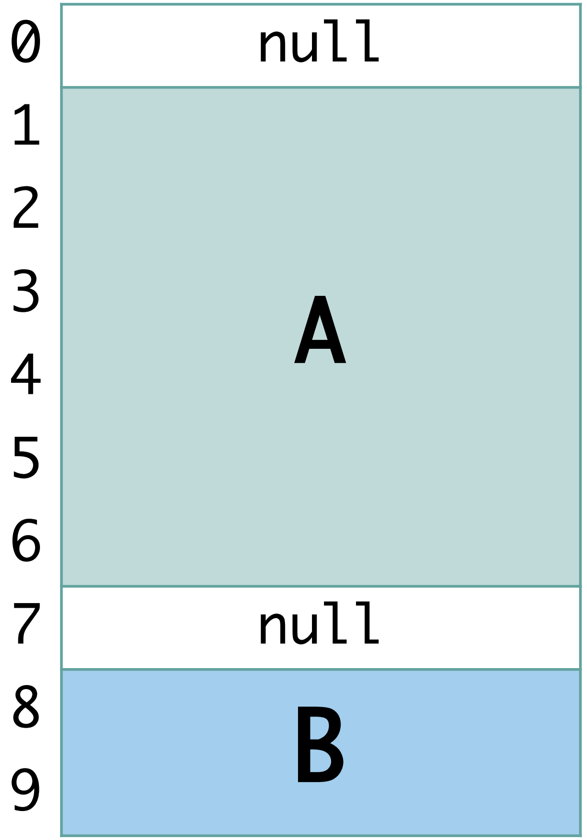
- A: size 5
- B: size 2

Probability that a new inserted key k has a bucket in

- cluster A? 5/10
- cluster B? 2/10

What happens after k is added?

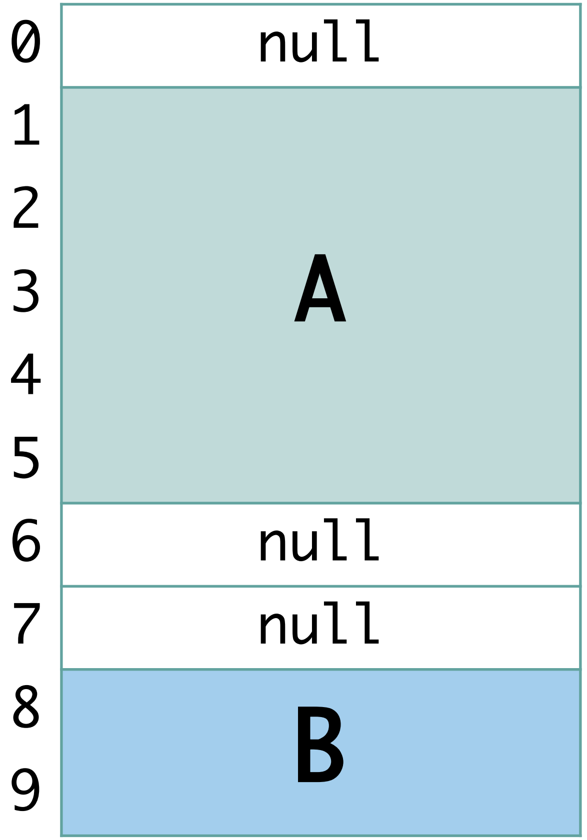
Cluster grows by 1 element



A PROBLEM: CLUSTERS

cluster = collection of consecutive occupied slots

What happens when I search for a key that has $h(k) = 1$, but k doesn't exist in the hash table?



WHY DO CLUSTERS MATTER?

α	1/4	1/2	2/3	3/4	9/10
successful	1.2	1.5	2.0	3.0	5.5
unsuccessful	1.4	2.5	5.0	8.5	50.5

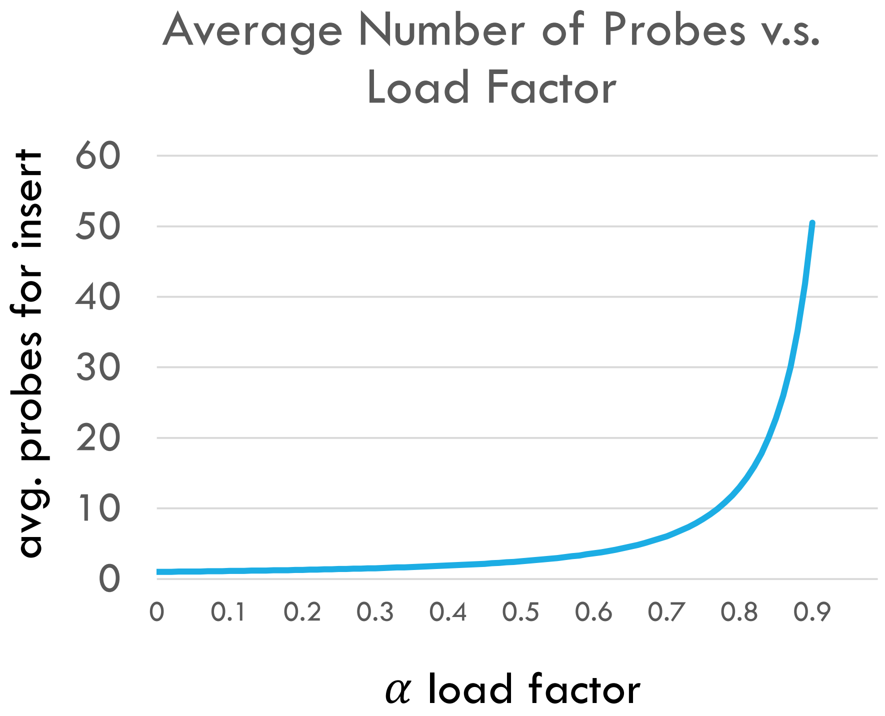
http://www.cs.cmu.edu/afs/cs/academic/class/15210-f13/www/lectures/lecture24.pdf

With table size m and $n = \alpha m$ keys, the average number of linear probes is:

$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$ for search hits

$\sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$ for search misses

Recall $\alpha \leq 1$ is the "load factor"



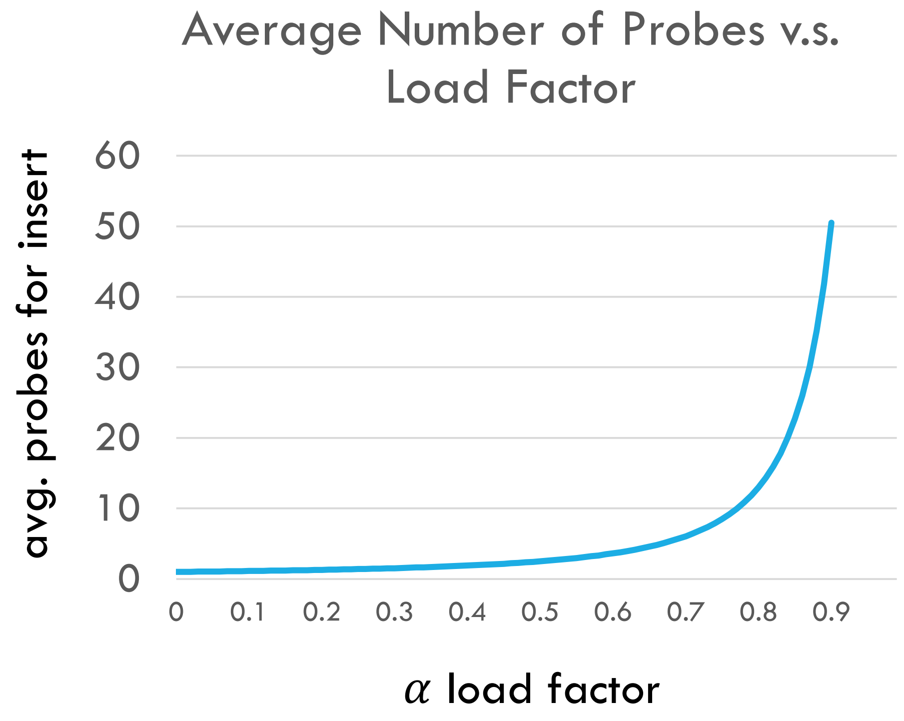
CAN WE REDUCE THE CLUSTERING EFFECT?

α	1/4	1/2	2/3	3/4	9/10
successful	1.2	1.5	2.0	3.0	5.5
unsuccessful	1.4	2.5	5.0	8.5	50.5

http://www.cs.cmu.edu/afs/cs/academic/class/15210-f13/www/lectures/lecture24.pdf

Ideas:

- Probe differently?
- Maintain a low load-factor α



OPEN ADDRESSING: QUADRATIC PROBING

Linear probing: index $i = (h(k) + \text{step} \times 1) \bmod m$

Quadratic probing: index $i = (h(k) + \text{step}^2) \bmod m$

Example:

- $h(k) = 3, m = 7$
- Step 0: $i = h(k) = 3$
- Step 1: $i = (h(k) + 1) \bmod 7 = 4$
- Step 2: $i = (h(k) + 4) \bmod 7 = 0$
- Step 3: $i = (h(k) + 9) \bmod 7 = 5$

Is this a good probing method?

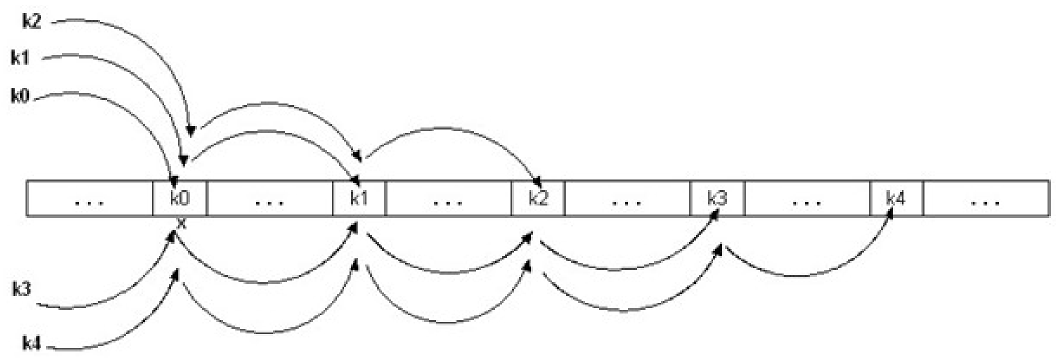
ONE PROBLEM: SECONDARY CLUSTERING

Milder form of the clustering problem.

Because: if two keys have the same probe position, their probe sequences are the same.

Clustering around different points (rather than the primary probe point)

index $i = (h(k) + \text{step}^2) \bmod m$



How many probe sequences can there be? m

ANOTHER PROBLEM: LET'S HEAD TO VISUALGO

What's going on?

$h(17) = 17 \, mod \, 7 = 3$ is already occupied by key 10

$(3 + 1^2) \, mod \, 7 = 4$ is already occupied by key 18

$(3 + 2^2) \, mod \, 7 = 0$ is already occupied by key 35

$(3 + 3^2) \, mod \, 7 = 5$ is already occupied by key 12

$(3 + 4^2) \, mod \, 7 = 5$ again is already occupied by key 12

$(3 + 5^2) \, mod \, 7 = 0$ again is already occupied by key 35

$(3 + 6^2) \, mod \, 7 = 4$ again is already occupied by key 18

$(3 + 7^2) \, mod \, 7 = 3$ again is already occupied by key 10

… infinite loop!

0	35
1	null
2	null
3	10
4	18
5	12
6	null

HOW TO FIX?

Idea: Show that the slots visited during the first m/2 probes are distinct

Theorem: if the load factor $\alpha = \frac{1}{2}$ and m is prime, then an empty slot will always be found via Quadratic Hashing.

Proof Sketch (by contradiction): Consider two probe locations $h(k) + x^2$ and $h(k) + y^2$ and $0 \leq x < y < \lceil m/2 \rceil$.

Suppose the locations are the same, but $x \neq y$.

$$h(k) + x^2 = h(k) + y^2 \pmod{m}$$

$$x^2 = y^2 \pmod{m}$$

$$x^2 - y^2 = 0 \pmod{m}$$

$$(x - y)(x + y) = 0 \pmod{m}$$

Since m is prime, either $(x - y)$ or $(x + y)$ are divisible by m. But since both $x - y$ and $x + y$ are less than m, they cannot be divisible by m. Contradiction.

Requires table be less than ½ full!

Poll Everywhere

https://bit.ly/2LvG9bq

DOUBLE HASHING

Use a second hashing:

index i = $(h_1(k) \; + \; \text{step} \times h_2(k)) \bmod m$

Avoids secondary clustering by providing more unique probing sequences.

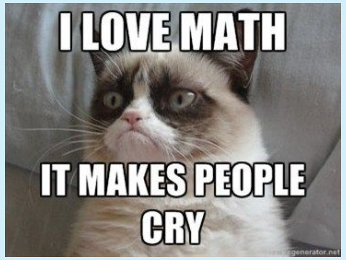
Up to how many unique indexing sequences does double hashing provide?

A. m

B. $2m$

C. m^2

D. 2^m

E. 

Poll Everywhere

https://bit.ly/2LvG9bq

DOUBLE HASHING

Use a second hashing:

index i $= (h_1(k)\ +\ \text{step} \times h_2(k)) \bmod m$

Avoids secondary clustering by providing more unique probing sequences.

Intuition:

- $\boldsymbol{h_1(k)}$ **provides good "random" base address**
- $\boldsymbol{h_2(k)}$ **provides good "random" sequence**

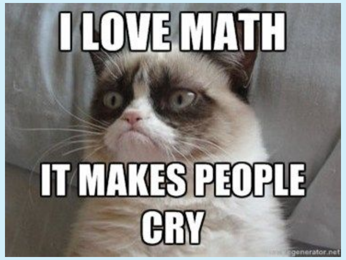
Up to how many unique indexing sequences does double hashing provide?

A. m

B. $2m$

C. $\boldsymbol{m^2}$

D. 2^m

E. 

DOUBLE HASHING

Use a second hashing:

index i = $(h_1(k) \; + \; \text{step} \times h_2(k)) \bmod m$

Avoids secondary clustering by providing **up to m^2** probing sequences.

To work:

- Needs careful choice for h_1 and h_2
- Make m prime and $h_2 < m$
- Also, $h_2(k) \neq 0$ **Why?**

One technique is to choose:

$$h_2 = (ak \; mod \; b) + 1$$

where $b \; < \; m$

Two integers are *relatively prime* if their greatest common divisor (GCD) is 1

DOUBLE HASHING HITS ALL BUCKETS

if $h_2(k)$ is relatively prime to m, double hashing hits all buckets

Proof Sketch (Contradiction):

Assume that this is not the case (some bucket is missed), then for some $i, j < m$ and $i \neq j$:

$$h_1(k) + ih_2(k) = h_1(k) + jh_2(k) \bmod m$$

$$ih_2(k) = jh_2(k) \bmod m$$

$$(i - j)h_2(k) = 0 \bmod m$$

Since $i, j < m$, $(i - j)$ is not divisible by m. But this implies $h_2(k)$ is divisible by m. However, this means that $h_2(k)$ is not relatively prime. Contradiction.

DOUBLE HASHING: WHAT ARE THE CONS?

Use a second hashing:

index i $= (h_1(k) + \text{step} \times h_2(k)) \bmod m$

Avoids secondary clustering by providing up to m^2 probing sequences.

Major con:

- Takes more time! Have to hash twice.

Poll Everywhere

https://bit.ly/2LvG9bq

OPEN ADDRESSING: COST OF INSERTS

What is the average cost of inserts (under SUHA) and $m = n$?

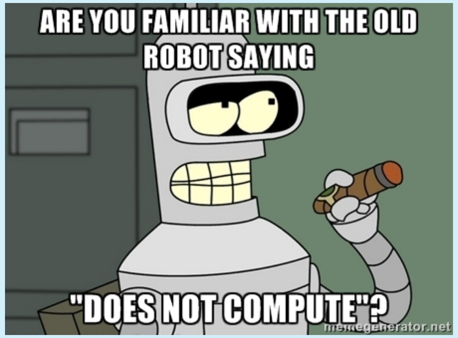
A. $O(m + n)$

B. $O(n \log m)$

C. $O(1)$

D. None of the above

E.



Poll Everywhere

https://bit.ly/2LvG9bq

OPEN ADDRESSING: COST OF INSERTS

What is the average cost of inserts (under SUHA) and $m = n$?

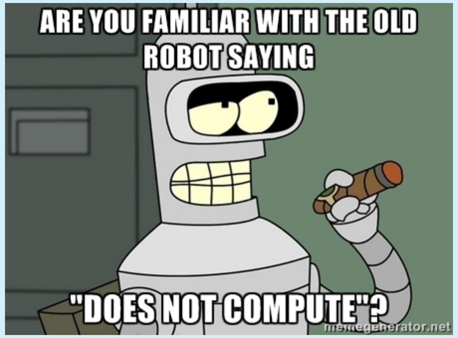
A. $O(m+n)$

B. $O(n \log m)$

C. $O(1)$

D. None of the above

E.



Why? I thought you said $O(1)$?

Note: The cost of an insert is the same as an unsuccessful search.

OPEN ADDRESSING PERFORMANCE (INSERTS)

Claim: For n items in a table of size m, assuming SUHA, the expected cost of an insert is $1/(1-\alpha)$ where $\alpha = \frac{n}{m}$

Proof Sketch:

p(1st bucket empty) $= \frac{m-n}{m} \equiv p$ $\boldsymbol{m-n}$ **free slots,** $\boldsymbol{m}$ **total**

p(2nd bucket empty | 1st probe failed) $= \frac{m-n}{m-1} \geq \frac{m-n}{m} = p$

p(3rd bucket empty | 1st & 2nd probe failed) $= \frac{m-n}{m-2} \geq \frac{m-n}{m} = p$

…

OPEN ADDRESSING PERFORMANCE (INSERTS)

Claim: For n items in a table of size m, assuming SUHA, the expected cost of an insert is $1/(1-\alpha)$ where $\alpha = \frac{n}{m}$

Proof Sketch:

At every trial, success with probability at least $p = \frac{m-n}{m} = 1 - \alpha$

$\mathbb{E}[\text{first success}] = \frac{1}{p} = \frac{1}{1-\alpha}$. So, time for insert is $O\left(\frac{1}{1-\alpha}\right)$

Note: For successful search $O\left(\frac{1}{\alpha}\left[1 + \log\frac{1}{1-\alpha}\right]\right)$

OPEN ADDRESSING

Linear Probing

$O\left(\frac{1}{(1-\alpha)^2}\right)$

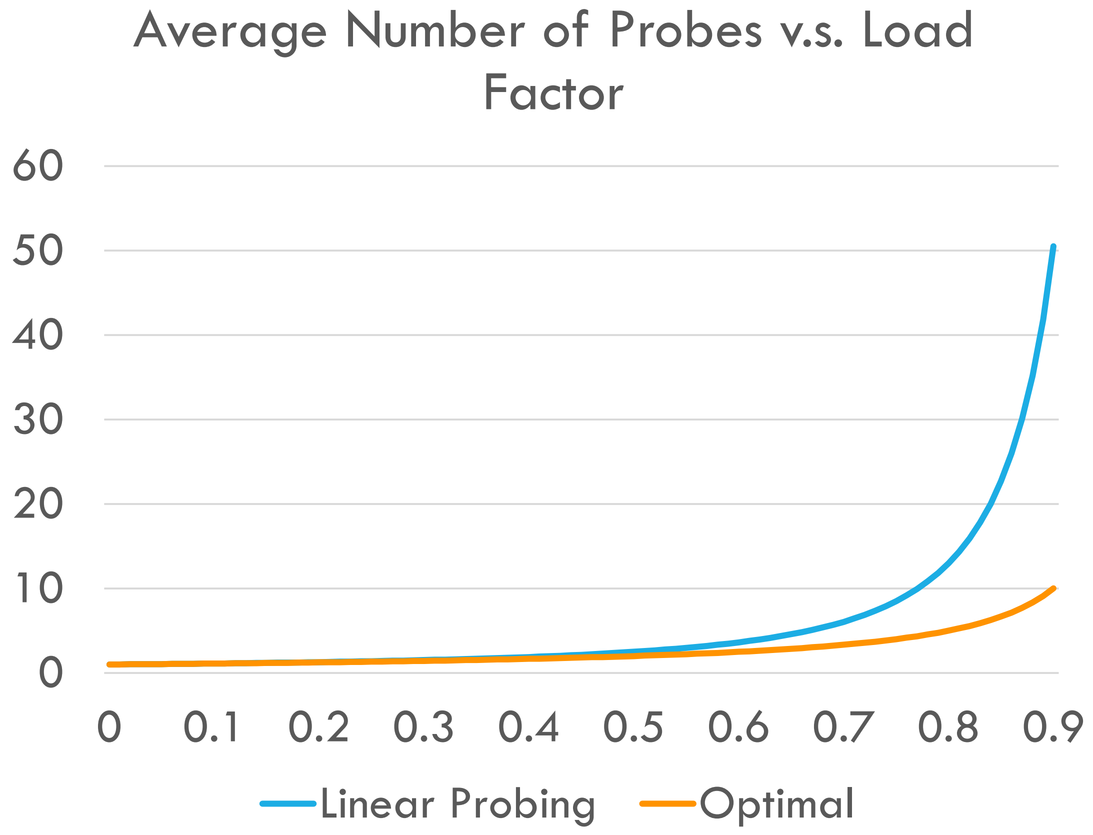
α	1/4	1/2	2/3	3/4	9/10
successful	1.2	1.5	2.0	3.0	5.5
unsuccessful	1.4	2.5	5.0	8.5	50.5

Optimal under SUHA

$O\left(\frac{1}{1-\alpha}\right)$

α	1/4	1/2	2/3	3/4	9/10
successful	1.2	1.4	1.6	1.8	2.6
unsuccessful	1.3	2.0	3.0	4.0	10.0

*double hashing comes close to this



http://www.cs.cmu.edu/afs/cs/academic/class/15210-f13/www/lectures/lecture24.pdf

Poll Everywhere

https://bit.ly/2LvG9bq

OPEN ADDRESSING: COST OF INSERTS

What is the average cost of inserts (under SUHA) and $\boxed{m = n?}$

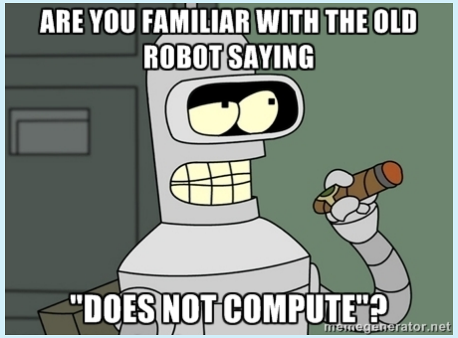
A. $O(m+n)$

B. $O(n \log m)$

C. $O(1)$

D. None of the above

E.



Why? I thought you said $O(1)$?

Poll Everywhere

https://bit.ly/2LvG9bq

OPEN ADDRESSING: COST OF INSERTS

What is the average cost of inserts (under SUHA) and $\alpha = 0.5$?

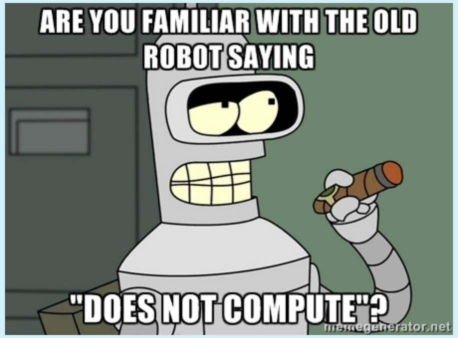
A. $O(m+n)$

B. $O(n \log m)$

C. $\boldsymbol{O}(\mathbf{1})$

D. None of the above

E.



$$O\left(\frac{1}{1-\alpha}\right) = O(2) = O(1)$$

CHAINING V.S. OPEN ADDRESSING

Open addressing:

- Better memory usage (cache performance)
- no references / dynamic memory allocation
- Linear addressing actually works well in practice

(Separate) Chaining:

- less sensitive to hash functions
- less sensitive to load factor (open addressing degrades past $\alpha = 0.7$)

Still, nice average O(1) properties

HMMM…

CHAINING V.S. OPEN ADDRESSING

Open addressing:

- Better memory usage (cache performance)
- no references / dynamic memory allocation
- Linear addressing actually works well in practice

(Separate) Chaining:

- less sensitive to hash functions
- less sensitive to load factor (open addressing degrades past $\alpha = 0.7$)

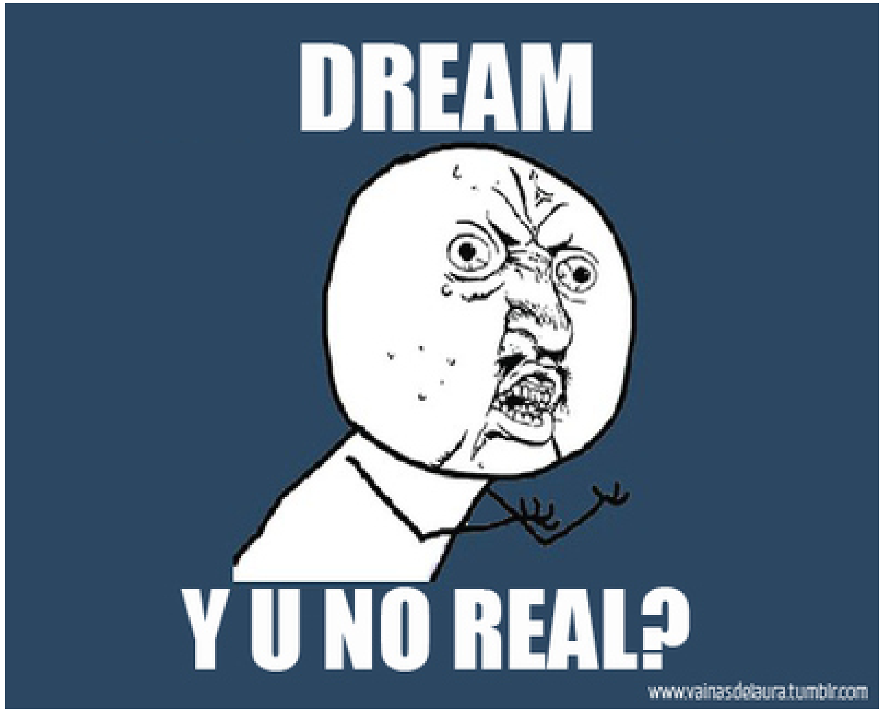
Still, nice average O(1) properties if assumptions met

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 $\in [0..m-1]$



TO BE CONTINUED …

In next lecture, we look at *designing* hashing functions…

LEARNING OUTCOMES

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

- Describe the **Symbol Table ADT**
- Explain the **Hash Table** Data Structure
- Analyze **the performance of the Hash Table**
- Describe the differences between the **Chaining** and **Open Addressing**

QUESTIONS?