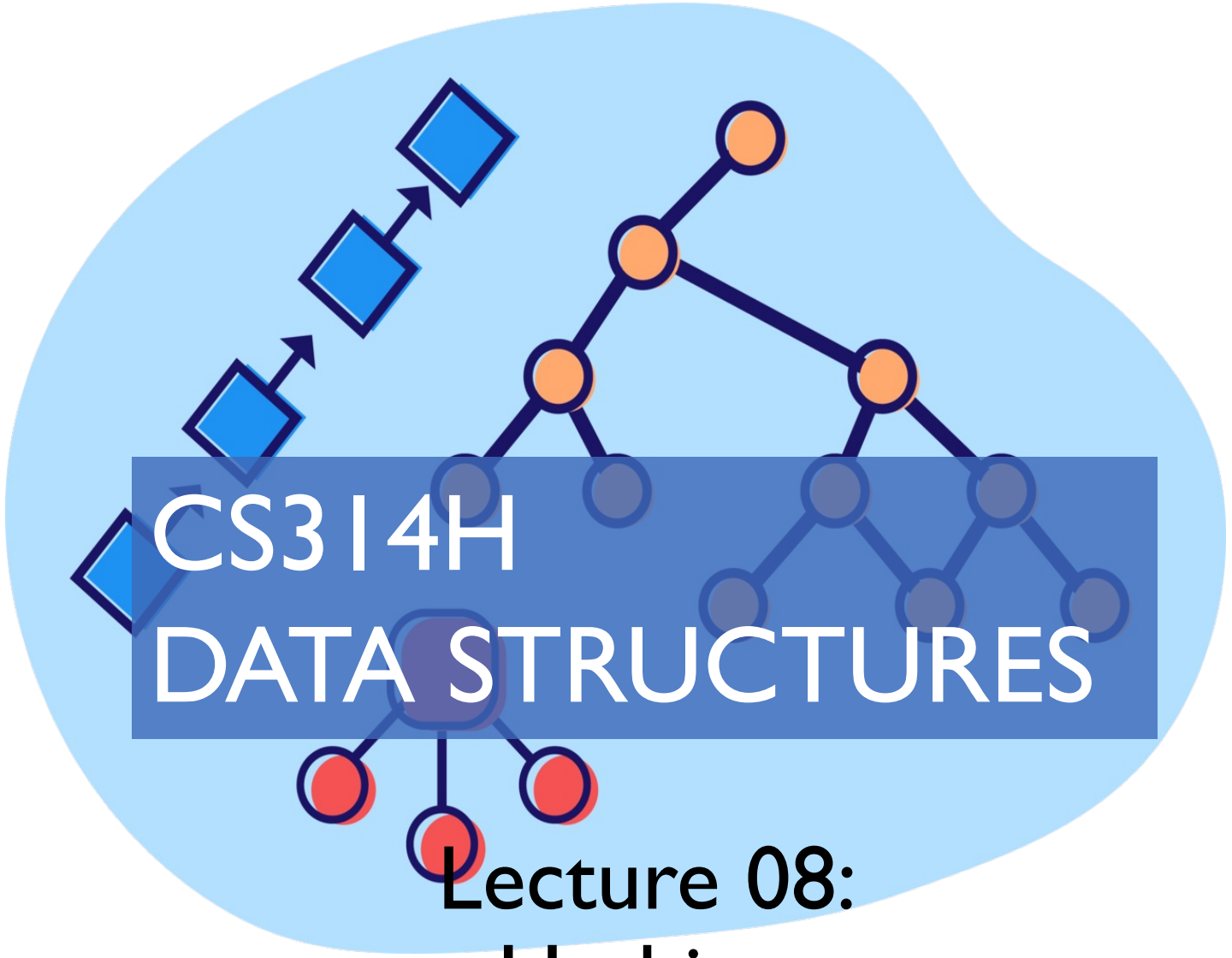


CS314H DATA STRUCTURES

Lecture 08: Hashing

Mikyung Han

Please, interrupt and ask questions AT ANY TIME !

Administrative

- **Prog 3 due Thursday 11:59 PM**
  - Your own critter for competition: Can work on it till CritterFest
- **CritterFest**
  - Fierce competition, party, food ☺
  - Time and date: TBD
- **Reading**
  - Ch 6-6.8 if you haven't done so
  - Ch 5

Outline

1. Administrative
2. Hash table Intro

Why (general-purpose) hash table?

- When order doesn't matter (no need to sort)
- Abstract Data Type
  - `void insert(key, value);`
  - `value find(key);`
  - `void remove(key);`

Wait, what is Abstract Data Type (ADT)?

- Defines a set of operations it supports (and a set of values it holds)
  - Stack: push pop peek
  - Queue: enqueue, dequeue
- Emphasizes the behavior of the data structure
- Does NOT dictate any particular implementation
- When it is actually implemented, we call it a _____

ADT defines what a data structure does,
while the actual data structure defines how it does it.

Why hash table?

- When order doesn't matter
- Abstract Data Type
  - `void insert(key, value);`
  - `value find(key);`
  - `void remove(key); //sometimes`
- We want O(1) performance!
  - Worst case O(n)
  - Average case O(1)

The hash table is the workhorse of computer systems

Notation

- U: Universe of all possible keys.
- K: Set of keys actually stored in the dictionary.
- |K| = n: the num of current elements stored
- m: size of the hash table

ADT

```
void insert(key, value);
value find(key);
void remove(key);
```

- Why not just use array?

In fact, it is called Direct-address Tables

- Direct-address Tables are ordinary arrays
- Store element with key k at index k
- All dictionary operations takes $O(1)$ time

- What is the big assumption here?
  - Key k is integer
  - The universe U is small

- What to do if k is NOT an integer?
- If U is too large what happens?

Notation

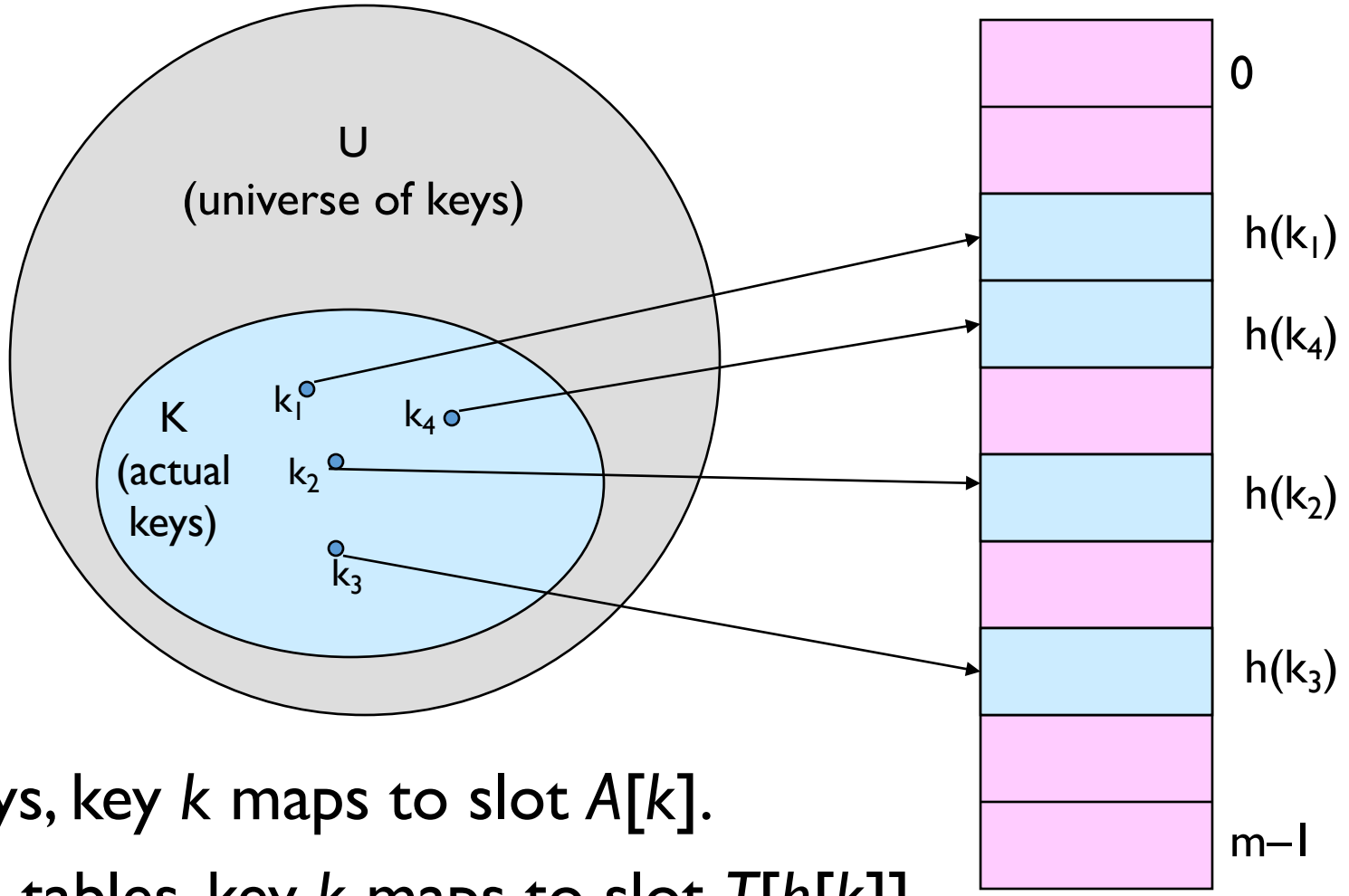
- U: Universe of all possible keys.
- K: Set of keys actually stored in the dictionary.
- |K| = n: the num of current elements stored
- m: size of the hash table

ADT

- `void insert(key, value);`
- `value find(key);`
- `void remove(key);`

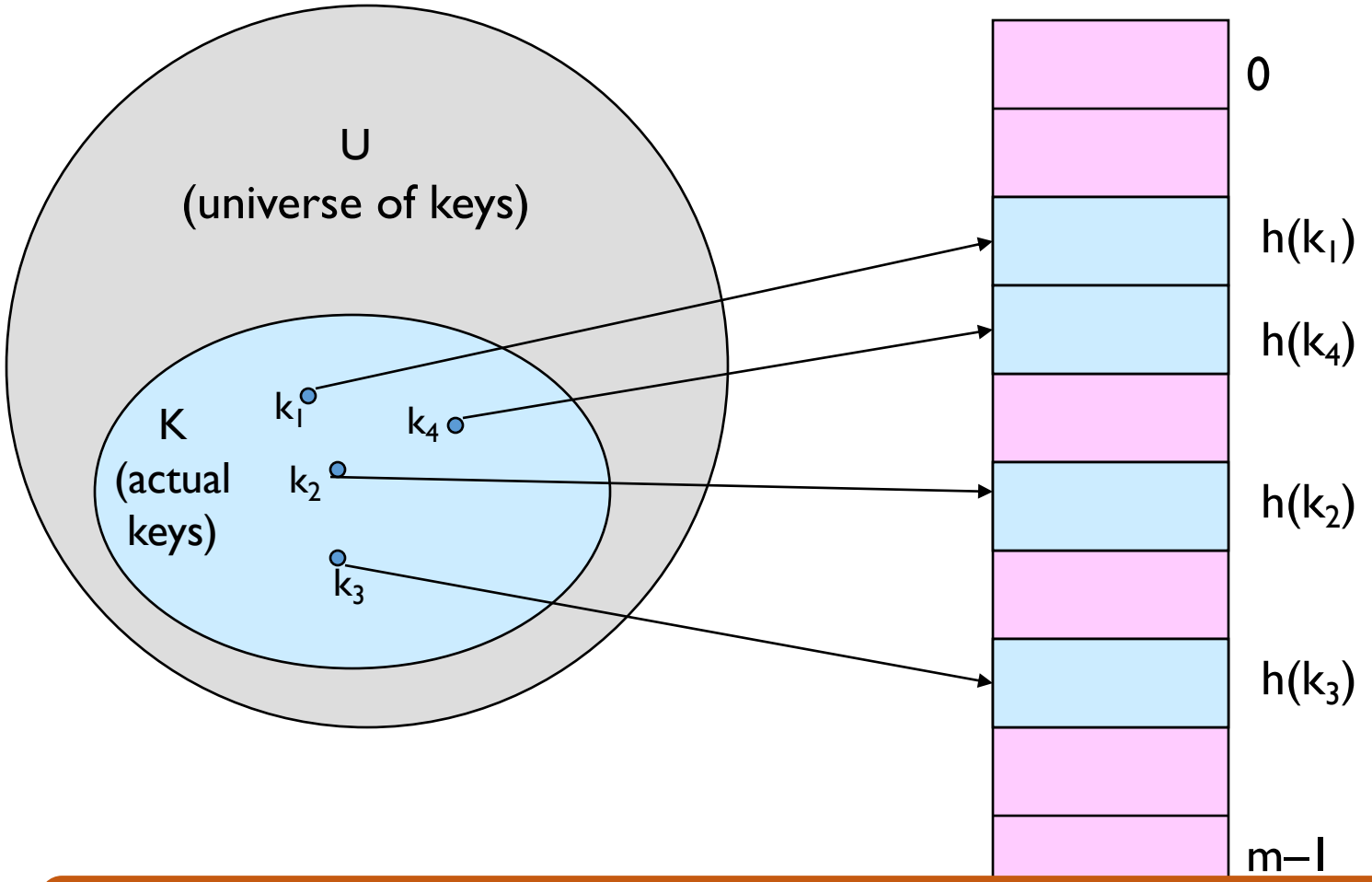
- We use should use hashing only if |U| >> |K|
- Use a table size proportional to |K|

Hash function *h* maps U to an index [0 .. m-1]



- With arrays, key *k* maps to slot *A*[*k*].
- With hash tables, key *k* maps to slot *T*[*h*[*k*]].
- *h*[*k*] is the *hash value* of key *k*.

Time complexity



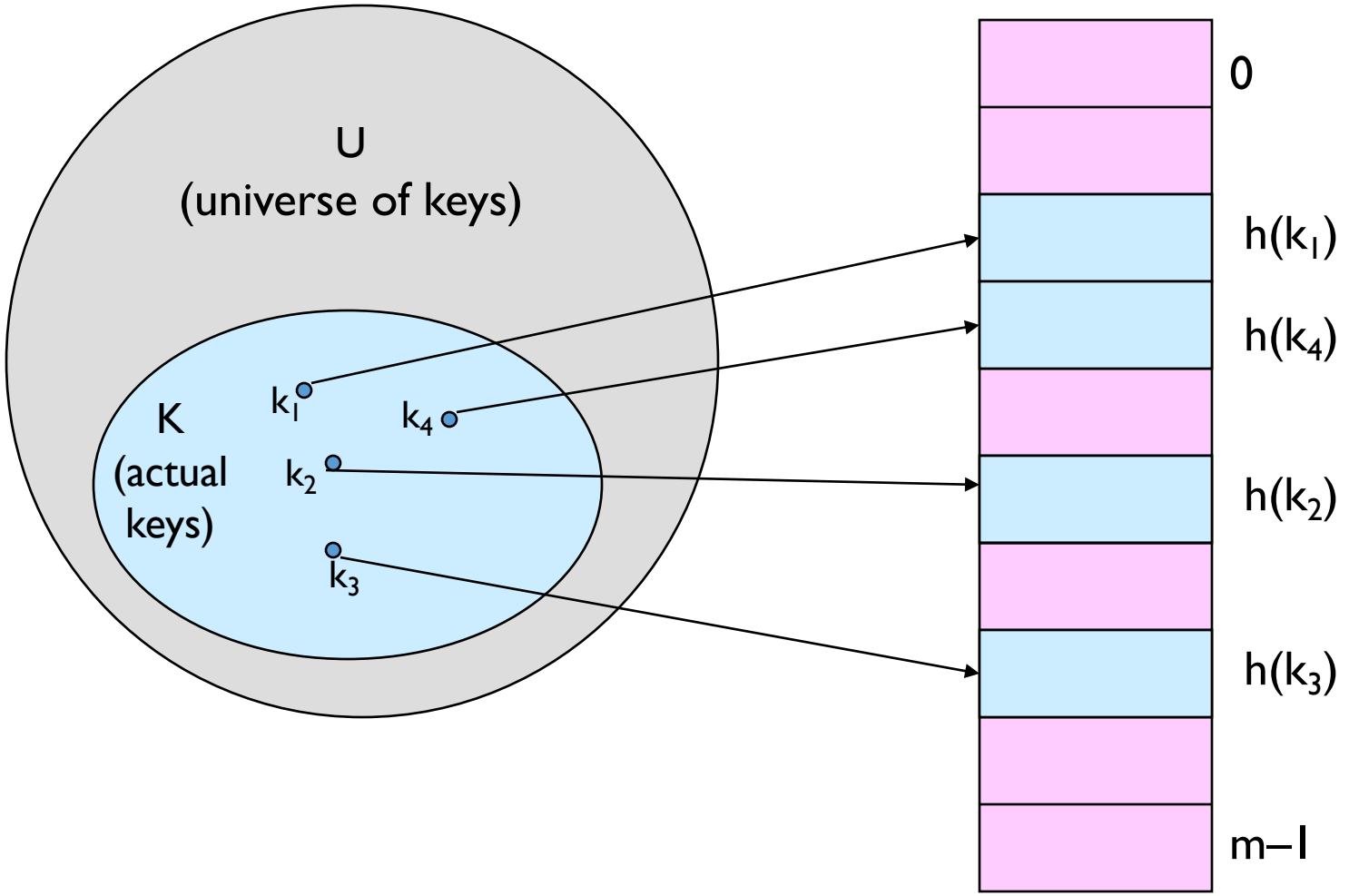
- Insert?
- Find?
- Delete?

All O(1) when life is good

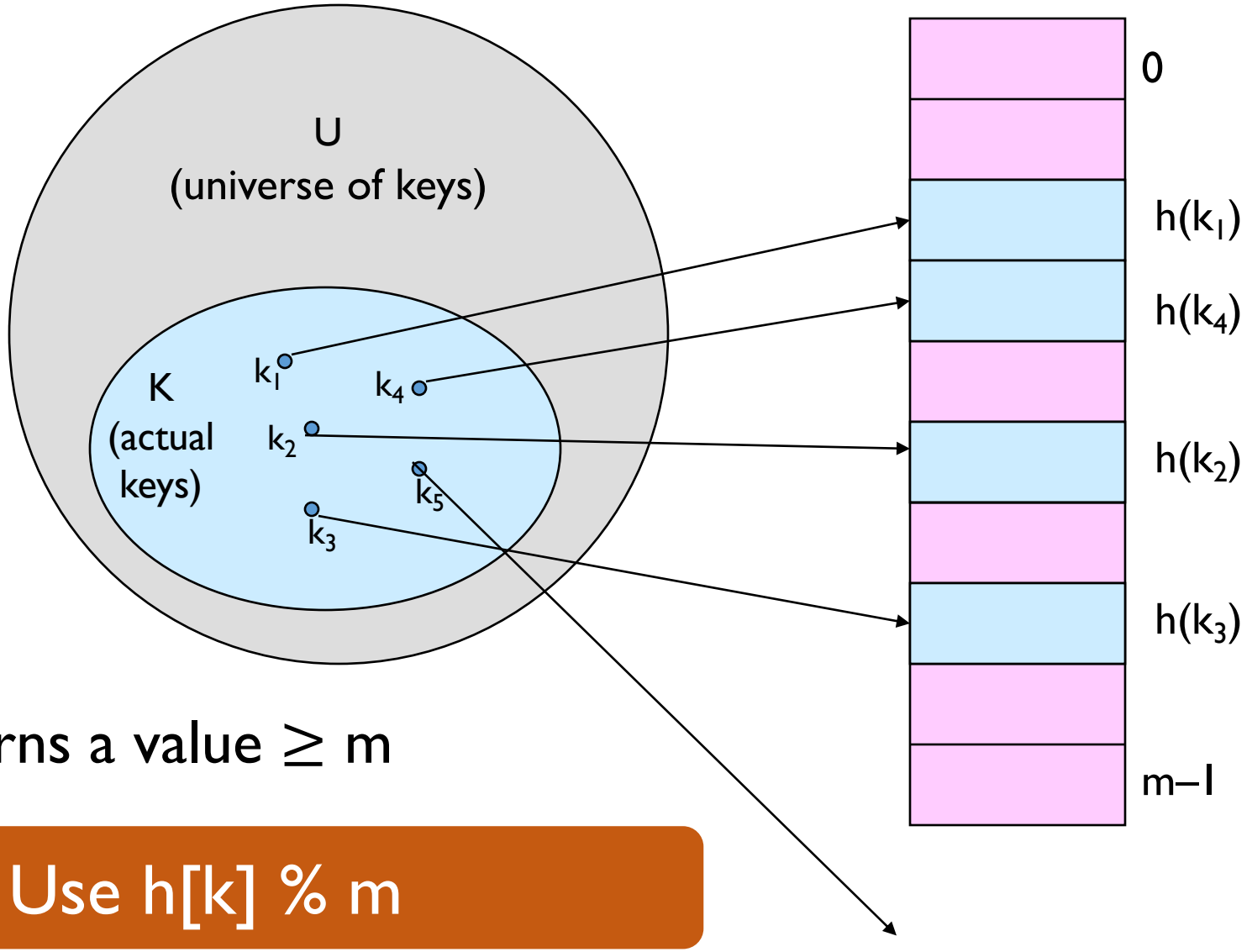
Outline

1. Administrative
2. Hash table Intro
3. Bad things can happen to good ~~people~~ hash table

What can go wrong with hashing?



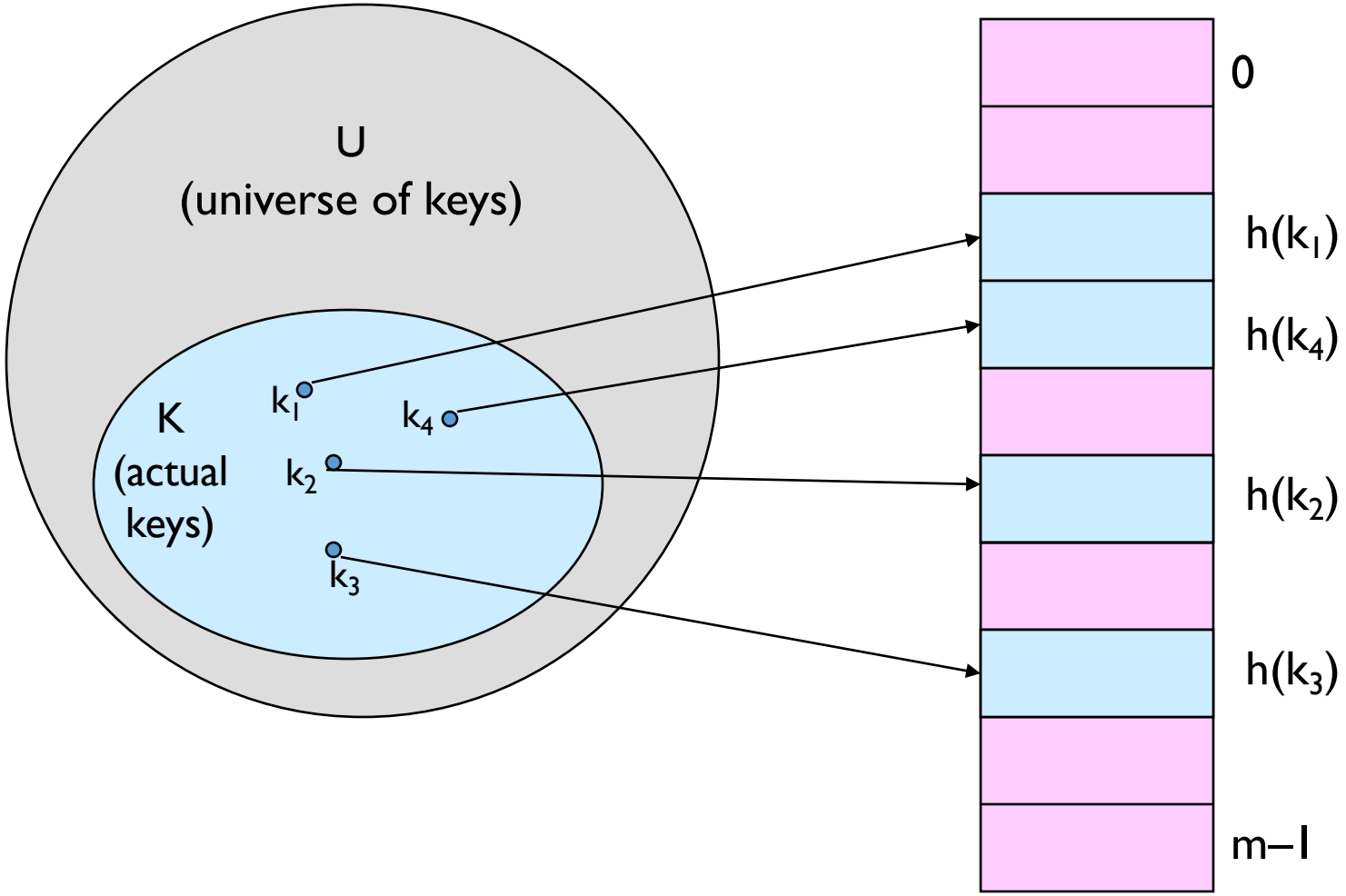
What can go wrong with hashing?



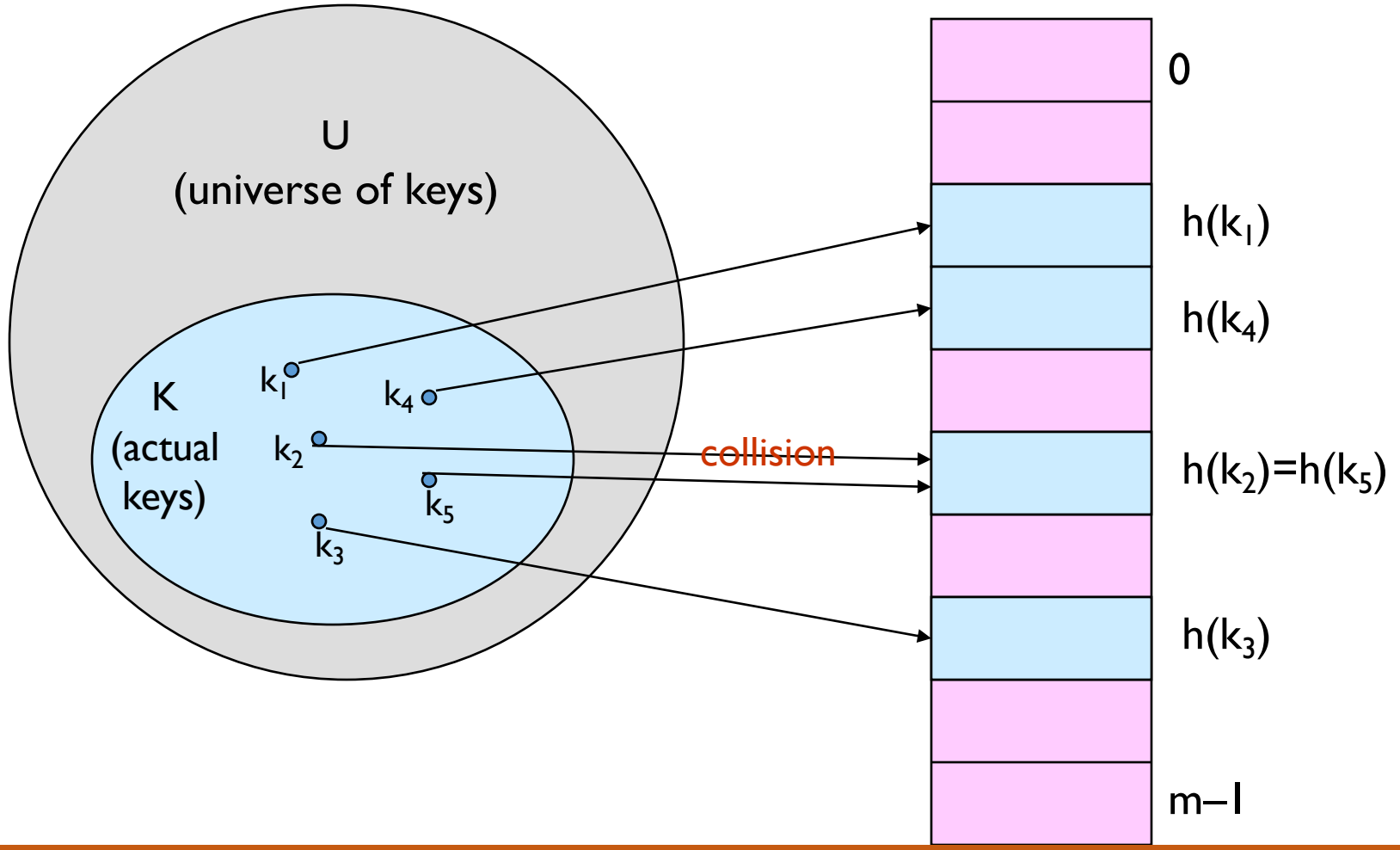
- Hash returns a value ≥ m

Use h[k] % m

What else can go wrong with hashing?



What else can go wrong with hashing?



Multiple keys can be mapped to the same slot causing collision!

What makes a good hash function?

MUST

- Deterministic: same input same output

GOOD

- Fast
- High uniformity: spread uniformly across the table

VERY GOOD

- Avalanche effect: small changes in input should cause large unpredictable changes in the output bits.

How do we deal with collisions?

There are multiple strategies:

- Separate Chaining
- Open Addressing
  - Linear Probing
  - Quadratic Probing
  - Double Hashing

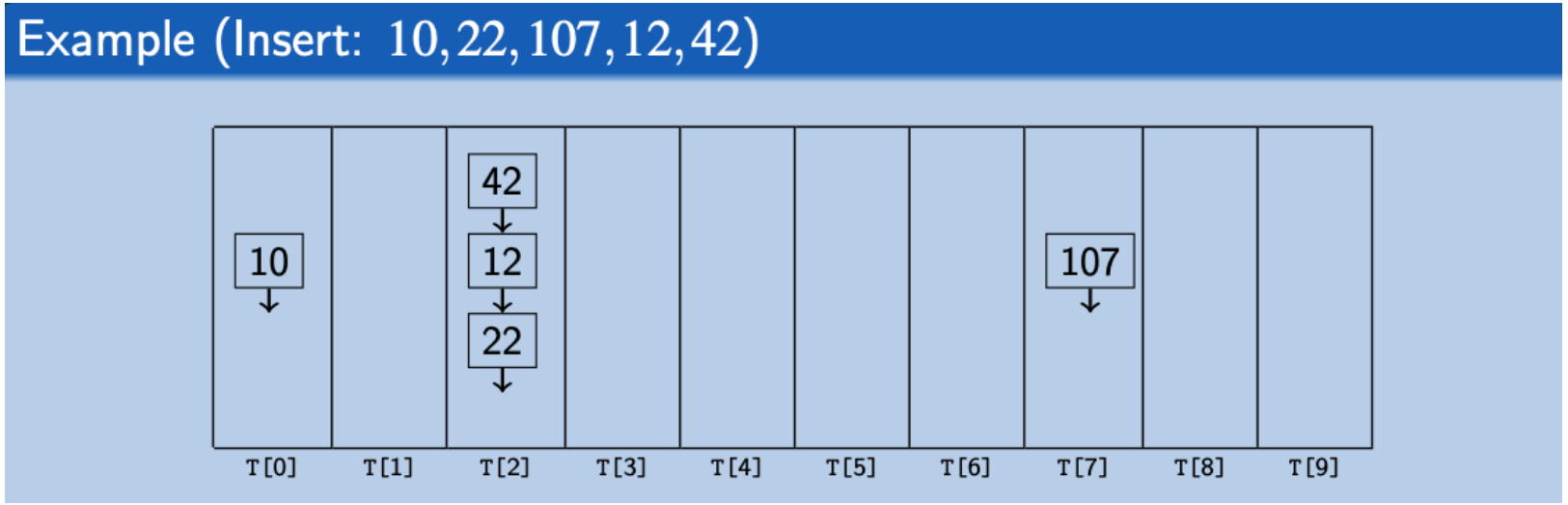
Outline

1. Administrative
2. Hash table Intro
3. Bad things can happen to good ~~people~~ hash table
4. Separate chaining

Separate chaining idea:

If we hash multiple keys to the same slot, store a LinkedList of them

- h(k) = k %10



- What is the worst-case time complexity for `insert(key)`?
- How about `find(key)` ?
- O(n): Absolute worst when using a horrible hash function h(k) = C
- What is the average case?

Load factor measures "how full" a hash table is

- $\lambda = \frac{n}{m}$
  - m is the size of the table
  - n = |K| is the number of items inserted

- Using separate chaining, what is the avg num of elements per slot?

Load factor measures "how full" a hash table is

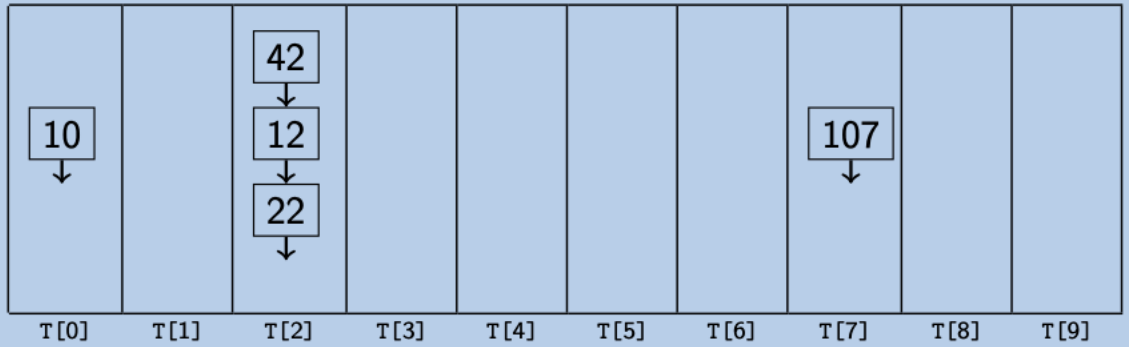
- $\lambda = \frac{n}{m}$
  - m is the size of the table
  - n = |K| is the number of items inserted

- Using separate chaining, the avg num of elements per slot is λ
- Each unsuccessful find compares against λ items
- Each successful find compares against λ items

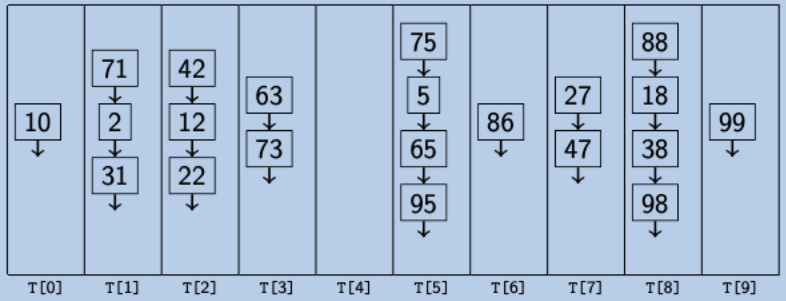
We want $\lambda \approx 1$ Why?

Load factor example

Example (What is the Load Factor?)



Example (What is the Load Factor?)

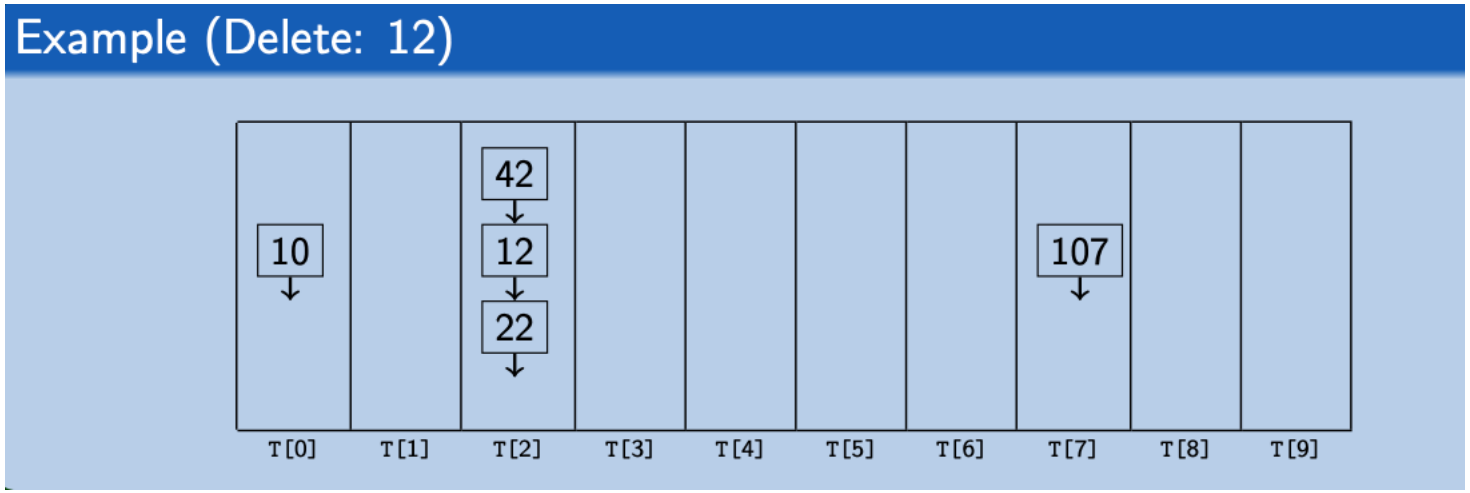


What is average case runtime of find?

Why average case is O(1)?

Separate chaining delete is simply a reverse of insert

- Remove from linked list!



- What is the worst-case time complexity for `delete(key)`?
- O(n): Absolute worst when using a horrible hash function h(k) = C
- Average case is O(1) with a good hash function

How to further improve performance?

- **Use better hash function**
  - MurmurHash, xxHash, CityHash
  - Speed, spread, avalanche effect, seed
  - Look to the experts, don't implement your own

- **Instead of using linked-list you can use fancier tree**
  - Java uses red-black tree (a balanced tree) if linked list size becomes > 8

What are the pros/cons of separate chaining?

Outline

1. Administrative
2. Hash table Intro
3. Bad things can happen to good ~~people~~ hash table
4. Separate chaining
5. Open addressing

Open Addressing resolves collisions by choosing a different location when the natural choice is full

Sure, different location OC! But what should be the criteria?

- We should be able to reproduce the path
  - Cannot be random. Still deterministic
- We want to use most of the spaces in the table
- We should avoid putting keys close together. Why?
  - More collisions means even more collisions!

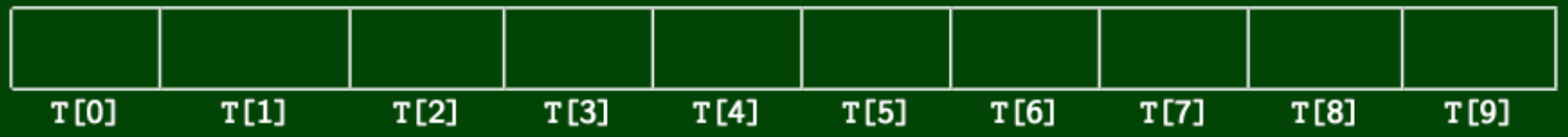
1st attempt: Linear Probing

Strategy #1: Linear Probing

```
1  i = 0;
2  while (index in use) {
3      try (h(key) + i) % |T|
4  }
```

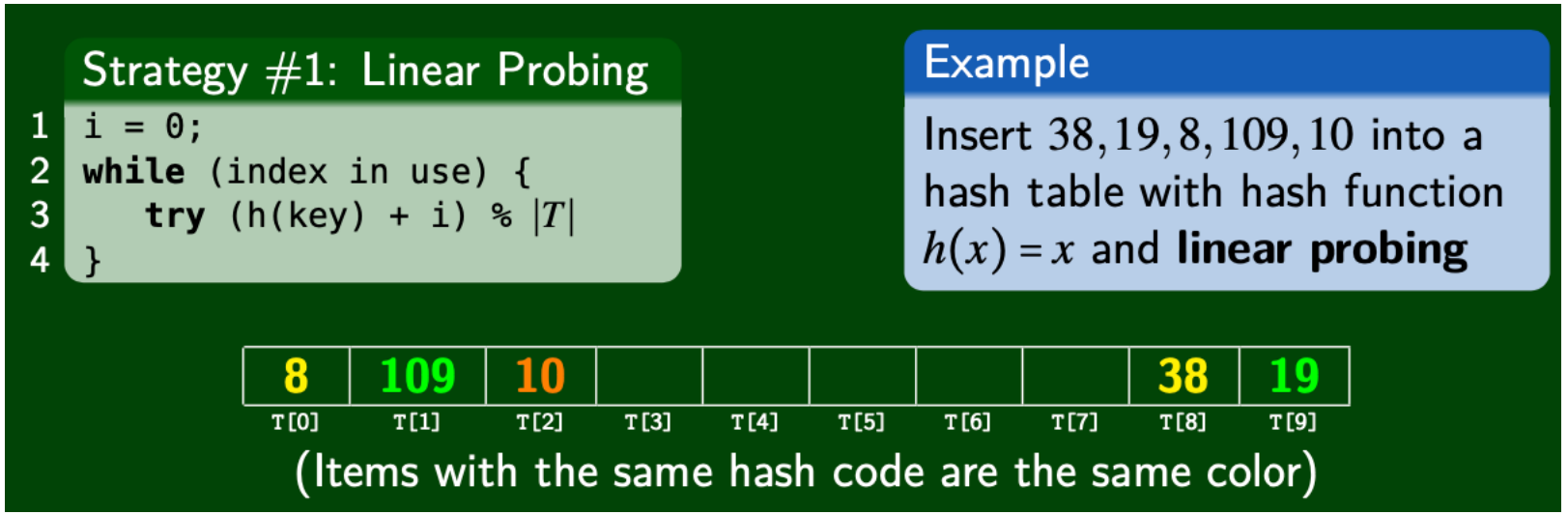
Example

Insert $38, 19, 8, 109, 10$ into a hash table with hash function $h(x) = x$ and **linear probing**



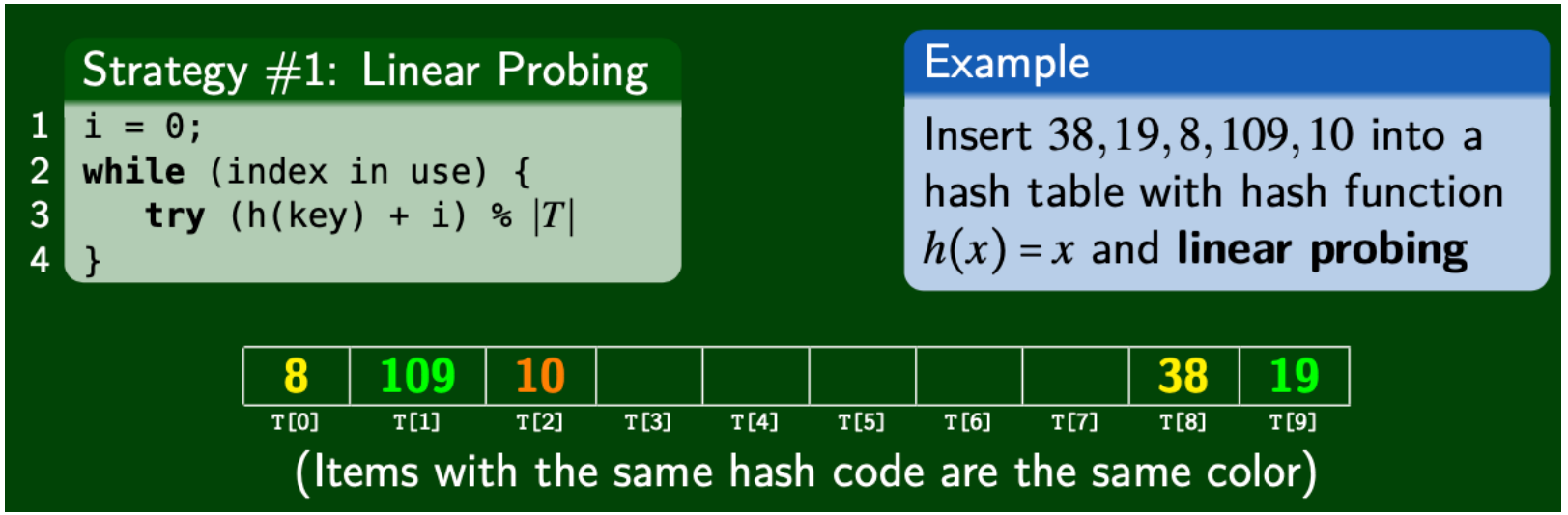
(Items with the same hash code are the same color)

Time complexity



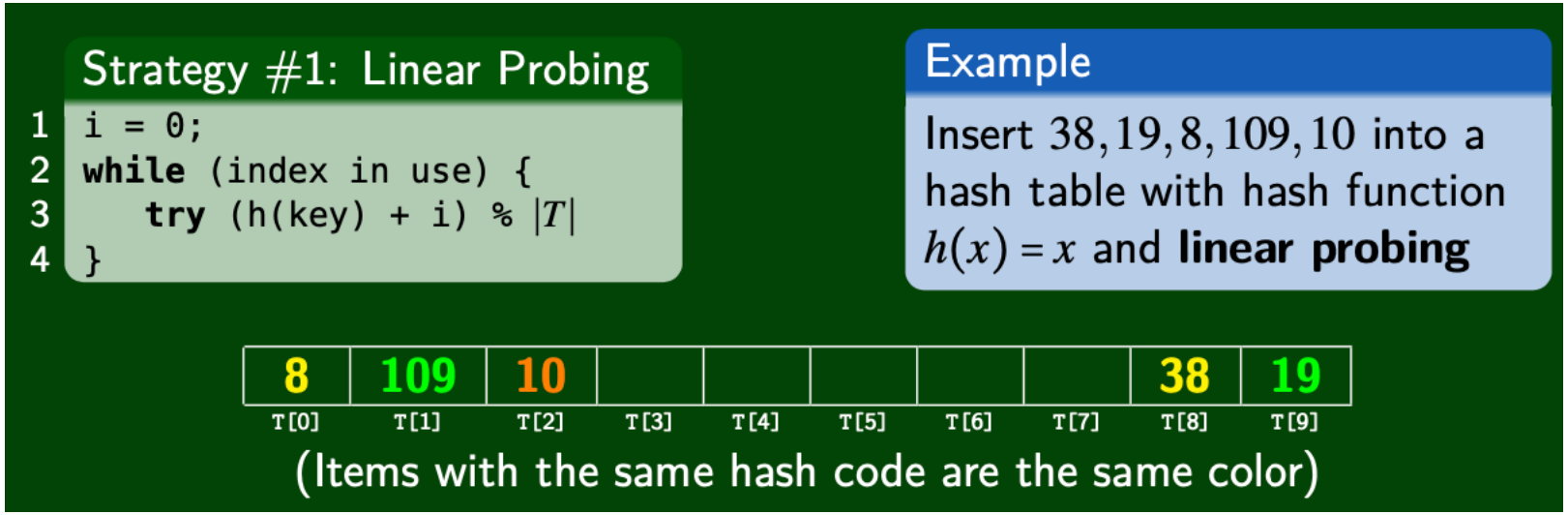
- Insert? What is the worst case of finding the next slot?
- Find?
- How to delete?

Time complexity



- Insert? What is the worst case of finding the next slot?
- Find?

Time complexity



- Insert? What is the worst case of finding the next slot?
- Find?
- Wait, can you delete? How?
  - What happens when you just delete 19 and try to find 109?
  - Lazy delete must be implemented

What is lazy delete?

Let's revisit the criteria and analyze Linear Probing

Thumbs up or down

- We should be able to reproduce the path
- We want to use most of the spaces in the table
- We should avoid putting keys close together

Linear probing leads to primary clustering

Primary clustering is
when different keys collide to form one big group

Which would have worse performance? (i.e., more # of probes?)

- Successful find
- Unsuccessful find

High-level intuition: Why unsuccessful search is far worse than successful search?

- In fact, exponentially worse

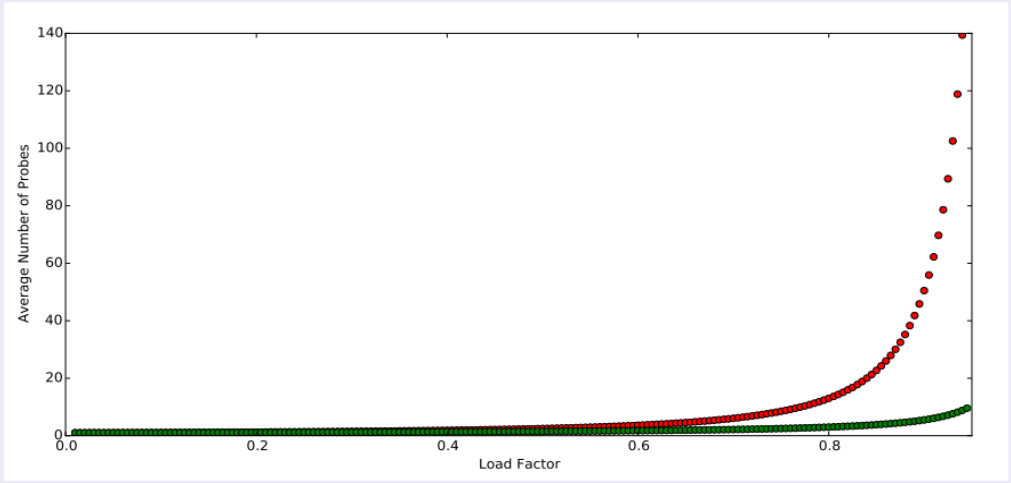
What is a good load factor?

Unsuccessful Search

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

Successful Search

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$$



- **Load factor = 0.5** (table half full)
  - Successful: $0.5 \times (1 + 2) = 1.5$ probes
  - Unsuccessful: $0.5 \times (1 + 4) = 2.5$ probes
- **Load factor = 0.9** (table almost full)
  - Successful: $0.5 \times (1 + 10) = 5.5$ probes
  - Unsuccessful: $0.5 \times (1 + 100) = 50.5$ probes (!!)

Mathematically….

- P_s is the average number of probes for successful find

$$P_s = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$$

Why 1 +

Why 1 / (1 - λ)

Why 1 / 2

Mathematically….

- P_u is the average number of probes for unsuccessful find

$$P_u = \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

Why square 1 / (1 - λ)^2?