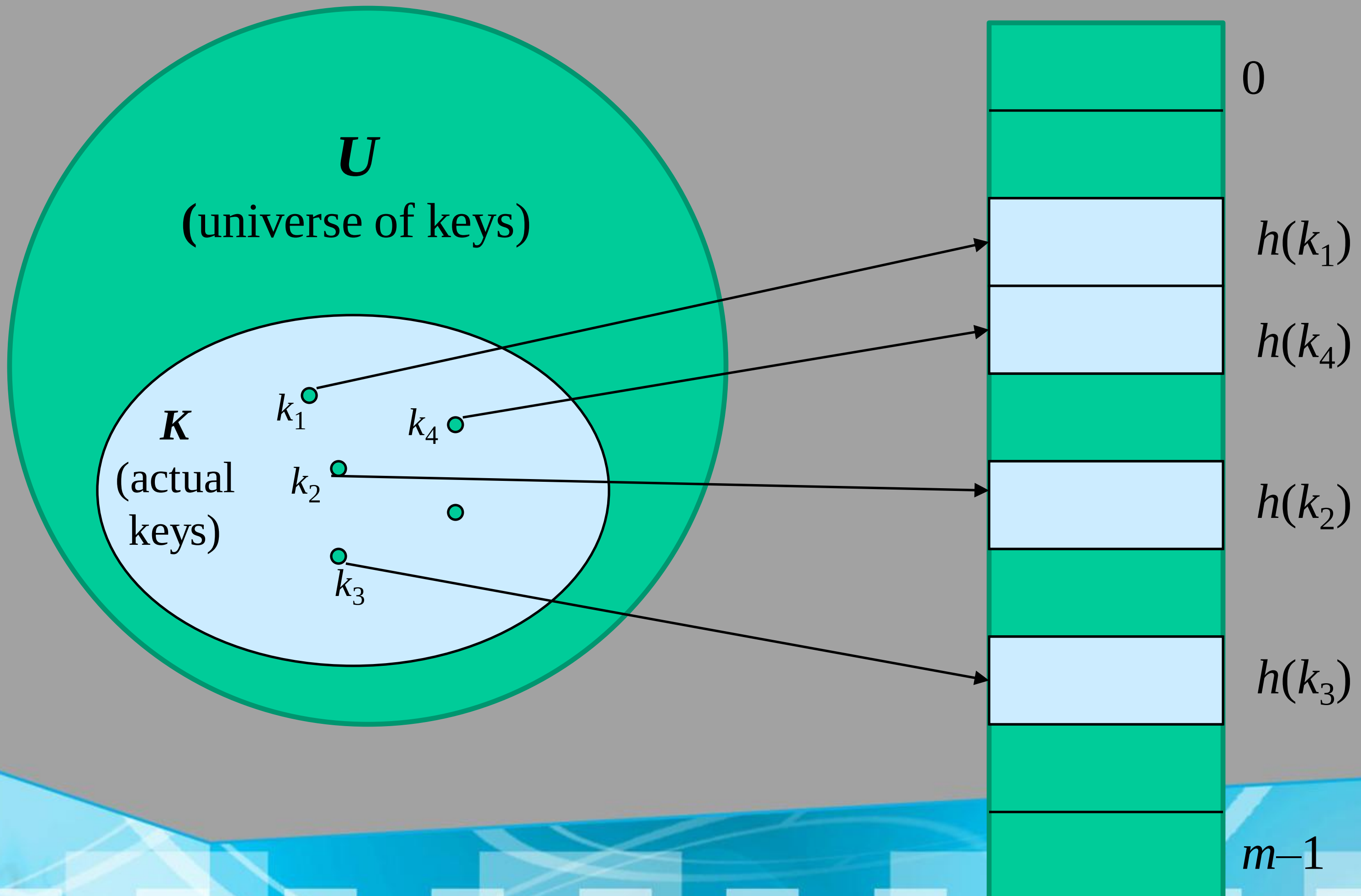
HASH TABLE

HASHING



HASHING

Hash function h: *Mapping from U to the slots of a hash table T[0..m–1].*

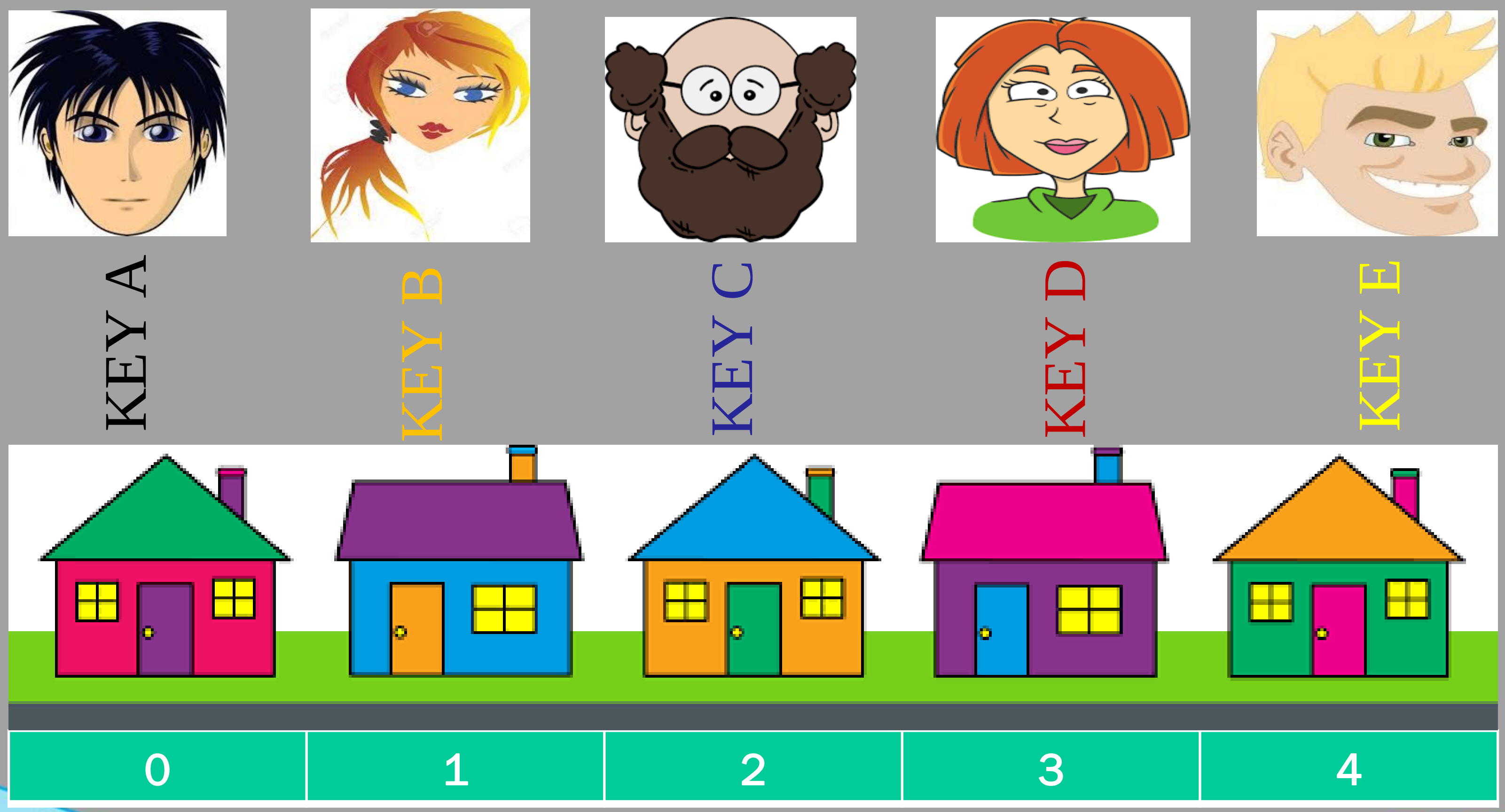
$$h : U \rightarrow \{0,1,\ldots, m-1\}$$

With arrays, key k maps to slot A[k].

With hash tables, key k maps or "hashes" to slot T[h[k]].

h[k] is the hash value of key k.

HASHING EXAMPLE



HASHING EXAMPLE

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

Figure 10.3: A lookup table with length 11 for a map containing entries (1,D), (3,Z), (6,C), and (7,Q).

Hash Tables

Notation:

- *U – Universe of all possible keys.*
- *K – Set of keys actually stored in the dictionary.*
- *$|K| = n$.*

When U is very large,

- *Arrays are not practical.*
- *$|K| << |U|$.*

Use a table of size proportional to $|K|$ – The hash tables.

- *However, we lose the direct-addressing ability.*
- *Define functions that map keys to slots of the hash table.*

HASING

Hash function h: *Mapping from U to the slots of a hash table T[0..m–1].*

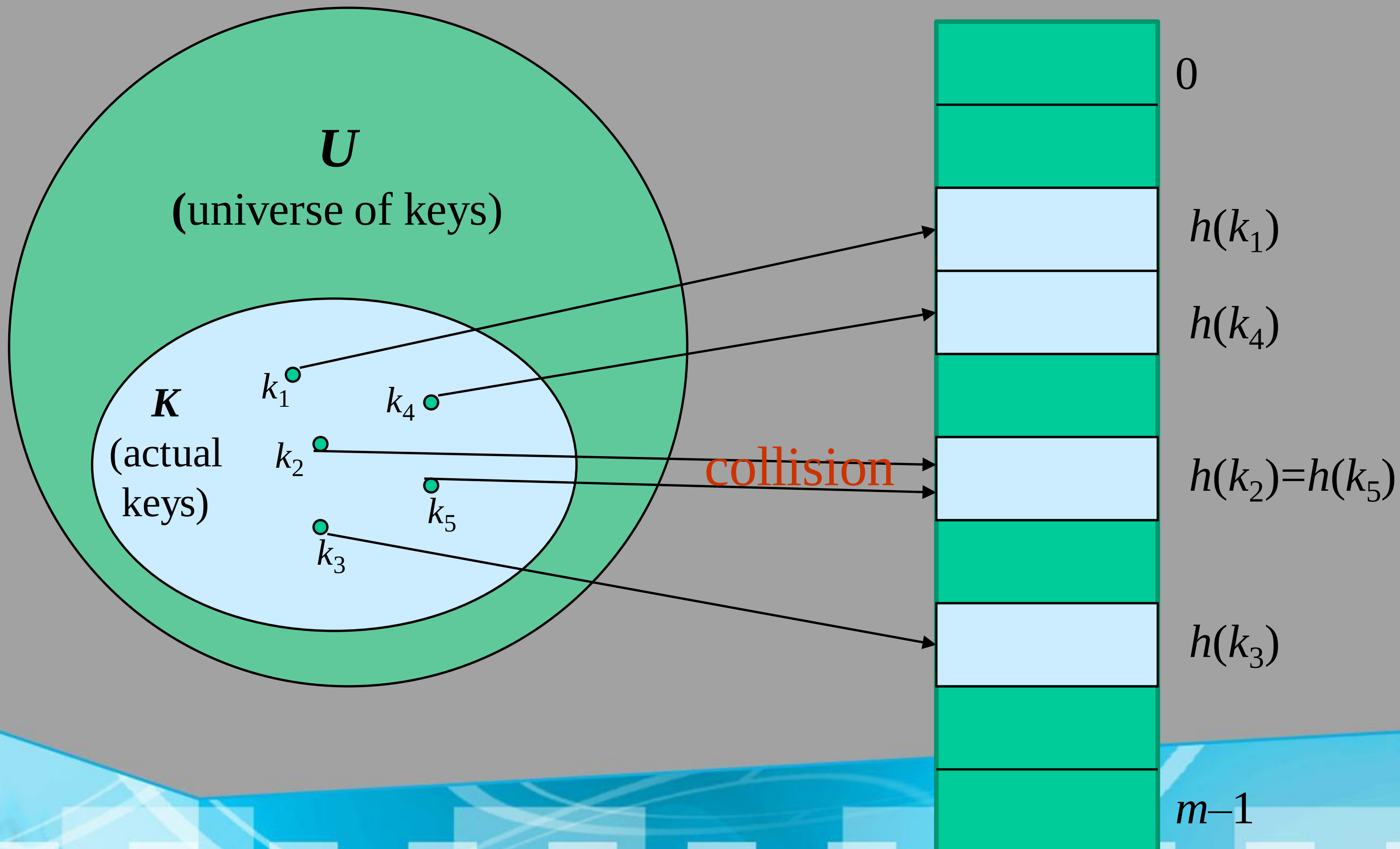
$$h : U \rightarrow \{0,1,\ldots, m-1\}$$

With arrays, key k maps to slot A[k].

With hash tables, key k maps or "hashes" to slot T[h[k]].

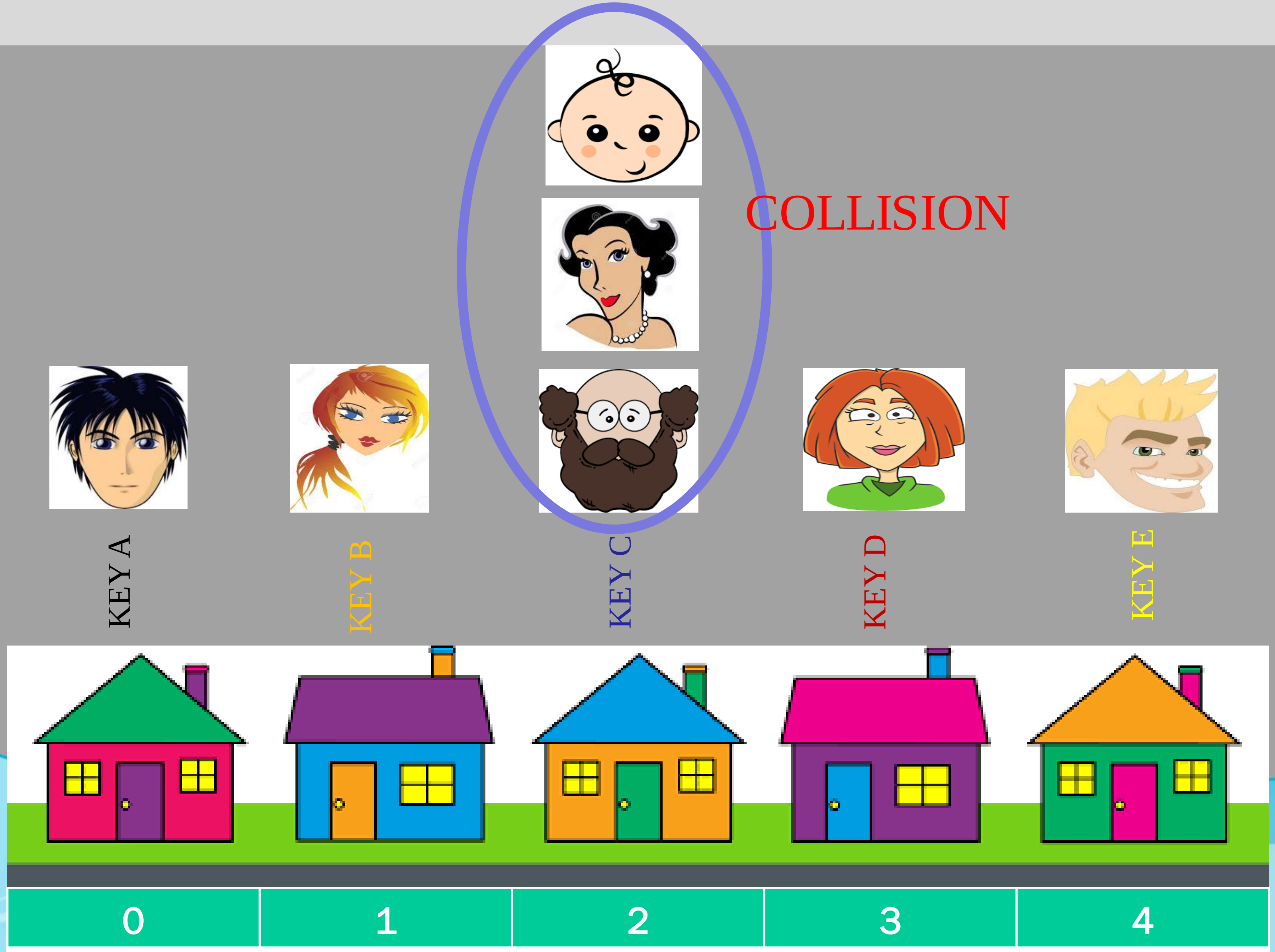
h[k] is the hash value of key k.

COLLISION



U
(universe of keys)
K
(actual keys)
k_1
k_4
k_2
k_5
k_3
collision
0
$h(k_1)$
$h(k_4)$
$h(k_2)=h(k_5)$
$h(k_3)$
$m–1$

COLLISION



COLLISION

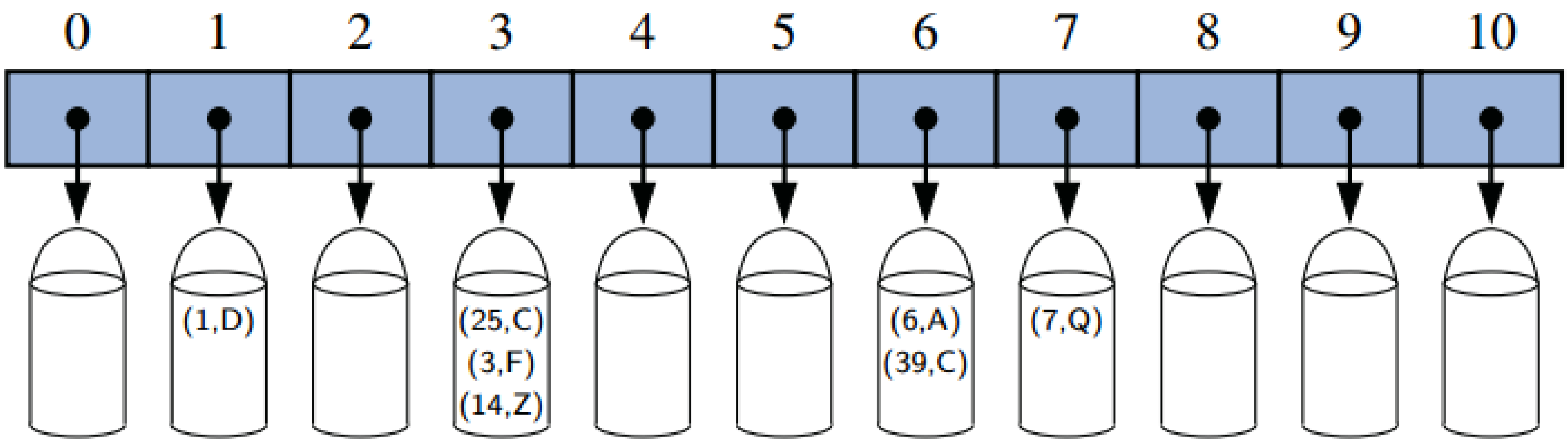


Figure 10.4: A bucket array of capacity 11 with entries (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function.

ISSUES WITH HASHING

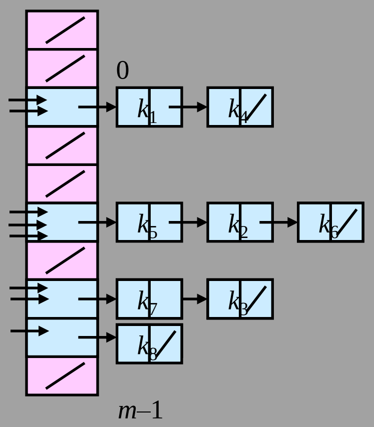
- *Multiple keys can hash to the same slot – collisions are possible.*
  - *Design hash functions such that collisions are minimized.*
  - *But avoiding collisions is impossible.*
    - *Design collision-resolution techniques.*
- *Search will cost Ө(n) time in the worst case.*
  - *However, all operations can be made to have an expected complexity of Ө(1).*

Methods of Resolution

Chaining:

Store all elements that hash to the same slot in a linked list.

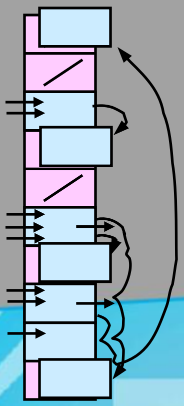
Store a pointer to the head of the linked list in the hash table slot.



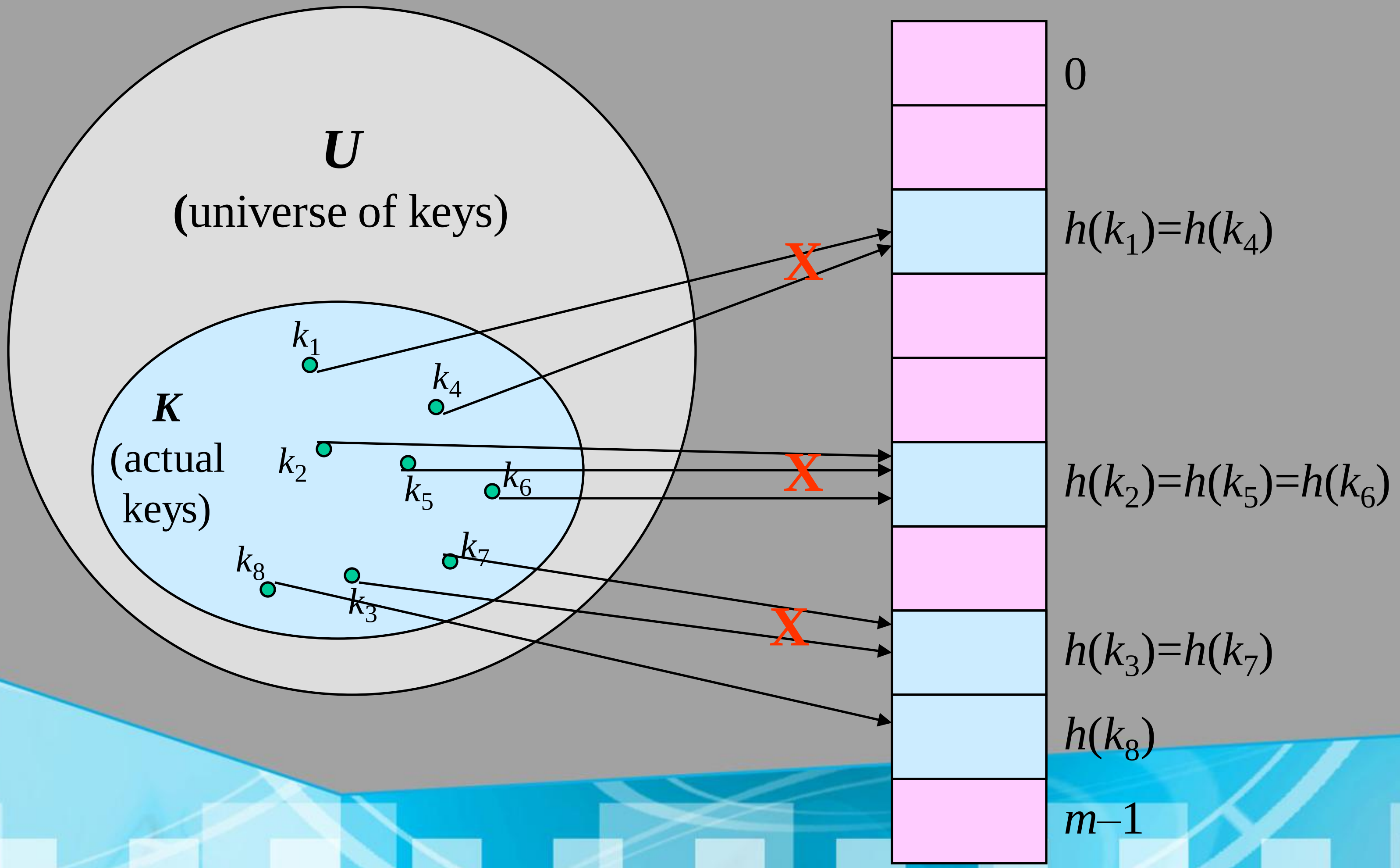
Open Addressing:

All elements stored in hash table itself.

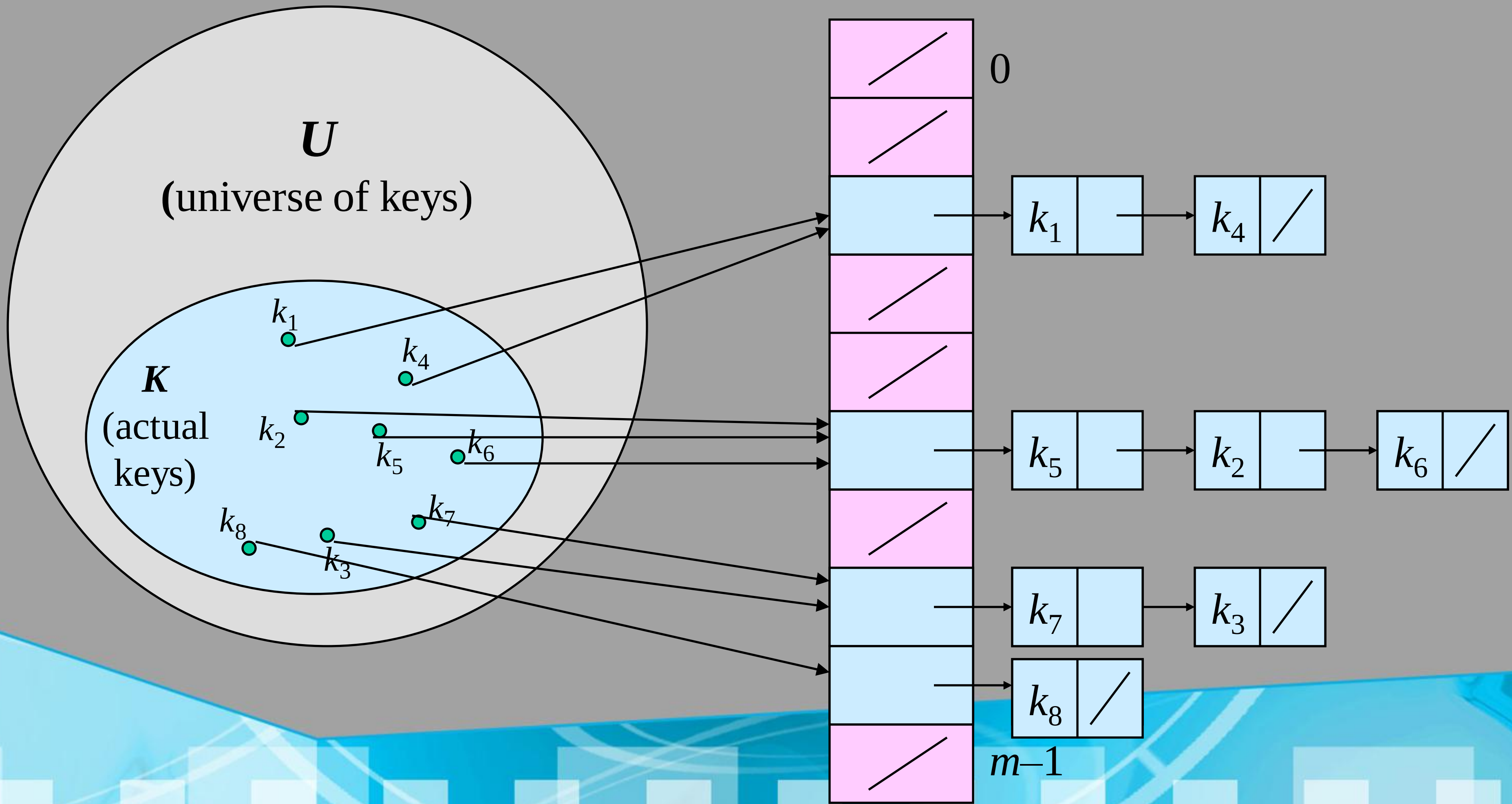
When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



Collision Resolution by Chaining



Collision Resolution by Chaining



Hashing with Chaining

Dictionary Operations:

Chained-Hash-Insert (T, x)

- *Insert x at the head of list T[h(key[x])].*
- *Worst-case complexity – O(1).*

Chained-Hash-Delete (T, x)

- *Delete x from the list T[h(key[x])].*
- *Worst-case complexity – proportional to length of list with singly-linked lists. O(1) with doubly-linked lists.*

Chained-Hash-Search (T, k)

- *Search an element with key k in list T[h(k)].*
- *Worst-case complexity – proportional to length of list.*

Analysis on Chained-Hash-Search

- *Load factor $\alpha = n/m$ = average keys per slot.*
  - *m – number of slots.*
  - *n – number of elements stored in the hash table.*
- *Worst-case complexity: $\Theta(n)$ + time to compute h(k).*

- *Average depends on how h distributes keys among m slots.*
- ***Assume***
  - *Simple uniform hashing.*
    - *Any key is equally likely to hash into any of the m slots, independent of where any other key hashes to.*
  - *O(1) time to compute h(k).*
- *Time to search for an element with key k is $\Theta(|T[h(k)]|)$.*
- *Expected length of a linked list = load factor = $\alpha = n/m$.*

Good Hash Functions

Satisfy the assumption of simple uniform hashing.

Not possible to satisfy the assumption in practice.

Often use heuristics, based on the domain of the keys, to create a hash function that performs well.

Regularity in key distribution should not affect uniformity. Hash value should be independent of any patterns that might exist in the data.

E.g. Each key is drawn independently from U according to a probability distribution P:

$$\sum_{k:h(k) = j} P(k) = 1/m \quad \text{for } j = 0, 1, \ldots, m-1.$$

An example is the division method.

Keys as Natural Numbers

Hash functions assume that the keys are natural numbers.

When they are not, have to interpret them as natural numbers.

Example: *Interpret a character string as an integer expressed in some radix notation. Suppose the string is CLRS:*

- *ASCII values: C=67, L=76, R=82, S=83.*
- *There are 128 basic ASCII values.*
- *So,* $CLRS = 67 \cdot 128^3 + 76 \cdot 128^2 + 82 \cdot 128^1 + 83 \cdot 128^0 = 141{,}764{,}947$.

Division Method

Map a key k into one of the m slots by taking the remainder of k divided by m. That is,

$$h(k) = k \bmod m$$

Example: $m = 31$ and $k = 78 \Rightarrow h(k) = 16$.

Advantage: *Fast, since requires just one division operation.*

Disadvantage: *Have to avoid certain values of m.*

- *Don't pick certain values, such as* $m=2^p$
- *Or hash won't depend on all bits of k.*

Good choice for m:

- *Primes, not too close to power of 2 (or 10) are good.*

Multiplication Method

If $0 < A < 1$, $h(k) = \lfloor m\,(kA \bmod 1) \rfloor = \lfloor m\,(kA - \lfloor kA \rfloor) \rfloor$

where kA mod 1 means the fractional part of kA, i.e., $kA - \lfloor kA \rfloor$.

Disadvantage: Slower than the division method.

Advantage: Value of m is not critical.

Typically chosen as a power of 2, i.e., $m = 2^p$*, which makes implementation easy.*

Example: $m = 1000$, $k = 123$, $A \approx 0.6180339887\ldots$

$$h(k) = \lfloor 1000(123 \cdot 0.6180339887 \bmod 1) \rfloor$$

$$= \lfloor 1000 \cdot 0.018169\ldots \rfloor = 18.$$

The MAD Method(***Multiply-Add-and-Divide***)

- $[(ai+b) \bmod p] \bmod N$
- N is the size of the bucket array
- p is a prime number larger than N, and a

  and b are integers chosen at random from the interval $[0, p-1]$, with $a > 0$. This compression function is chosen in order to eliminate repeated patterns in the set of hash codes and get us closer to having a “good” hash function, that is, one such that the probability any two different keys collide is $1/N$. This good behavior would be the same as we would have if these keys were “thrown” into A uniformly at random.

MAP _ DEFINITION

A set is a collection that lets you quickly find an existing element.

A map stores key/value pairs. You can find a value if you provide the key.

For example, you may store a table of employee records, where the keys are the employee IDs and the values are Employee objects.