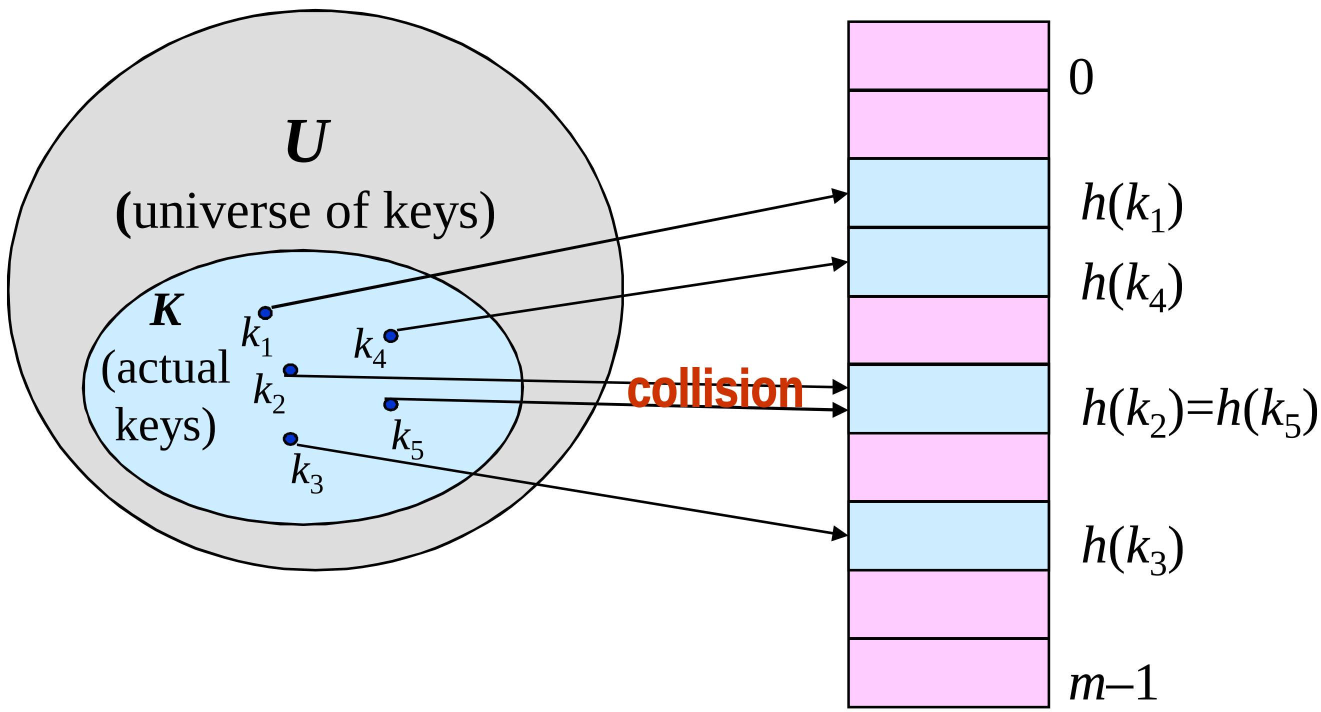
Hash Tables


U
(universe of keys)
K
(actual
keys)
k1
k2
k3
k4
k5
collision
0
h(k1)
h(k4)
h(k2)=h(k5)
h(k3)
m–1

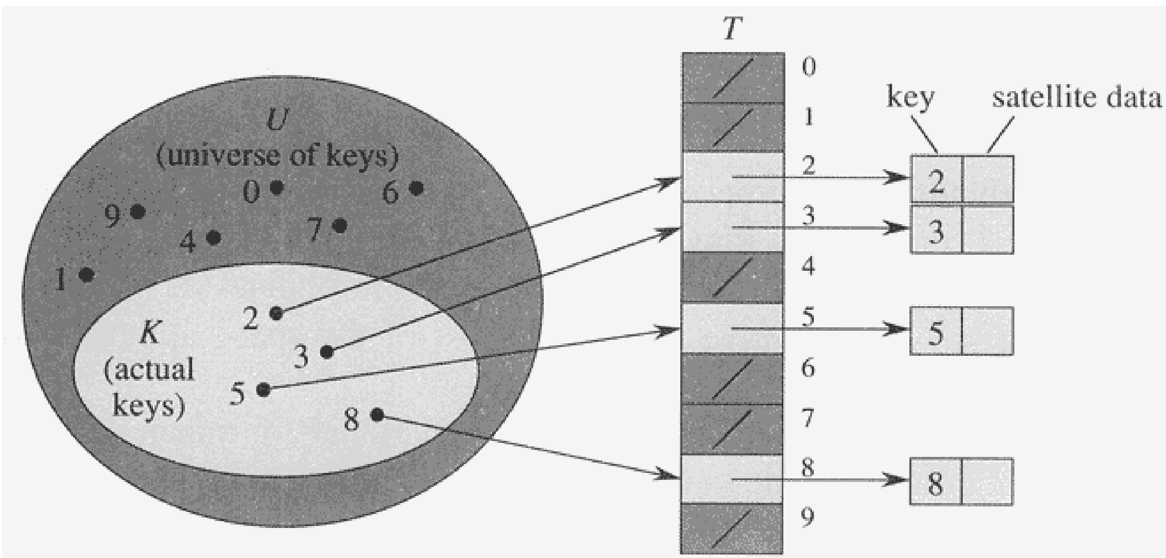
Dictionary

- **Dictionary:**
  - Dynamic-set data structure for storing items indexed using *keys*.
  - Supports operations Insert, Search, and Delete.
  - Applications:
    - Symbol table of a compiler.
    - Memory-management tables in operating systems.
    - Large-scale distributed systems.

- **Hash Tables:**
  - Effective way of implementing dictionaries.
  - Generalization of ordinary arrays.

Direct-Address Tables

- Direct-address Tables are ordinary arrays
- Facilitate direct addressing
  - Element whose key is k is obtained by indexing into the k^{th} position of the array
- Applicable when we can afford to allocate an array with one position for every possible key
  - i.e. when the universe of keys U is small
- Dictionary operations can be implemented to take $O(1)$ time



Direct-Address Tables

- Suppose:
  - The range of keys is 0..m-1
  - Keys are distinct
- The idea:
  - Set up an array T[0..m-1] in which
    - T[i] = x if $x \in T$ and key[x] = i
    - T[i] = NULL otherwise
  - This is called a *direct-address table*
    - Operations take O(1) time !
    - *So what's the problem?*

Direct-Address Tables

- Direct addressing works well when the range *m* of keys is relatively small
- But what if the keys are 32-bit integers?
  - Problem 1: direct-address table will have 2^{32} entries, more than 4 billion
  - Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be
- Solution: map keys to smaller range 0..*m*-1
- This mapping is called a *hash function*

Direct-Address Tables

Direct-Address-Search(T, k)
        return T[k]

Direct-Address-Insert(T, x)
        T[key[x]] $\leftarrow$ x

Direct-Address-Delete(T, x)
        T[key[x]] $\leftarrow$ NIL

We could use a direct-address table to implement caller-id, with the phone numbers as keys.

Time Analysis:

Space Analysis:

Hash Tables

- Motivation: symbol tables
  - A compiler uses a *symbol table* to relate symbols to associated data
    - Symbols: variable names, procedure names, etc.
    - Associated data: memory location, call graph, etc.
  - For a symbol table (also called a *dictionary*), we care about search, insertion, and deletion
  - We want these to be fast, but don't care about sorted order

- The structure we will use is a *hash table*
  - Supports all the above in O(1) expected time !

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.

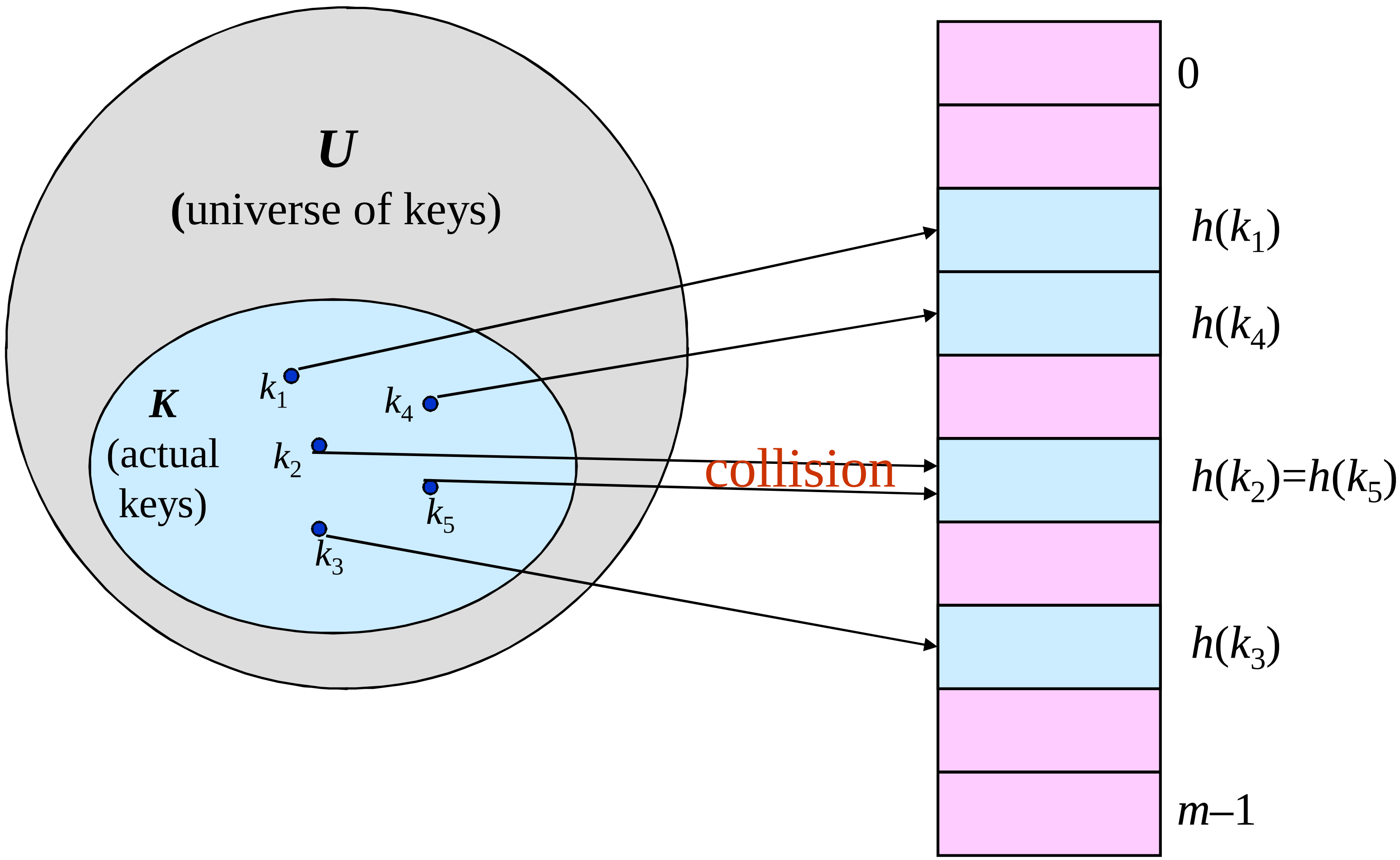
Hashing

- 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



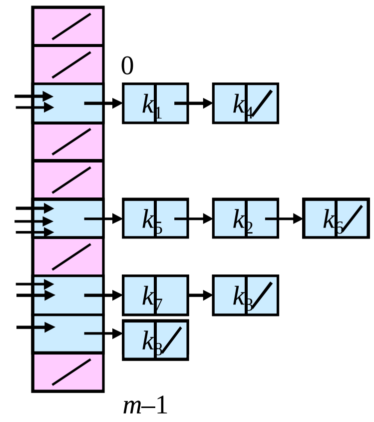
U
(universe of keys)
K
(actual
keys)
k_1
k_4
k_2
k_5
k_3
collision
0
$h(k_1)$
$h(k_4)$
$h(k_2)=h(k_5)$
$h(k_3)$
$m-1$

Issues with Hashing

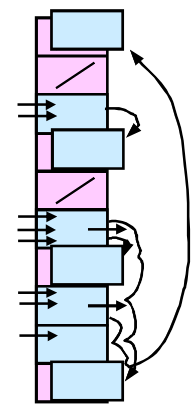
- Multiple keys can hash to the same slot – collisions are possible.
  - Design hash functions such that collisions are minimized.
  - But avoiding collisions is impossible.
    - Design collision-resolution techniques.
- Search will cost $\Theta(n)$ time in the worst case.
  - However, all operations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

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.



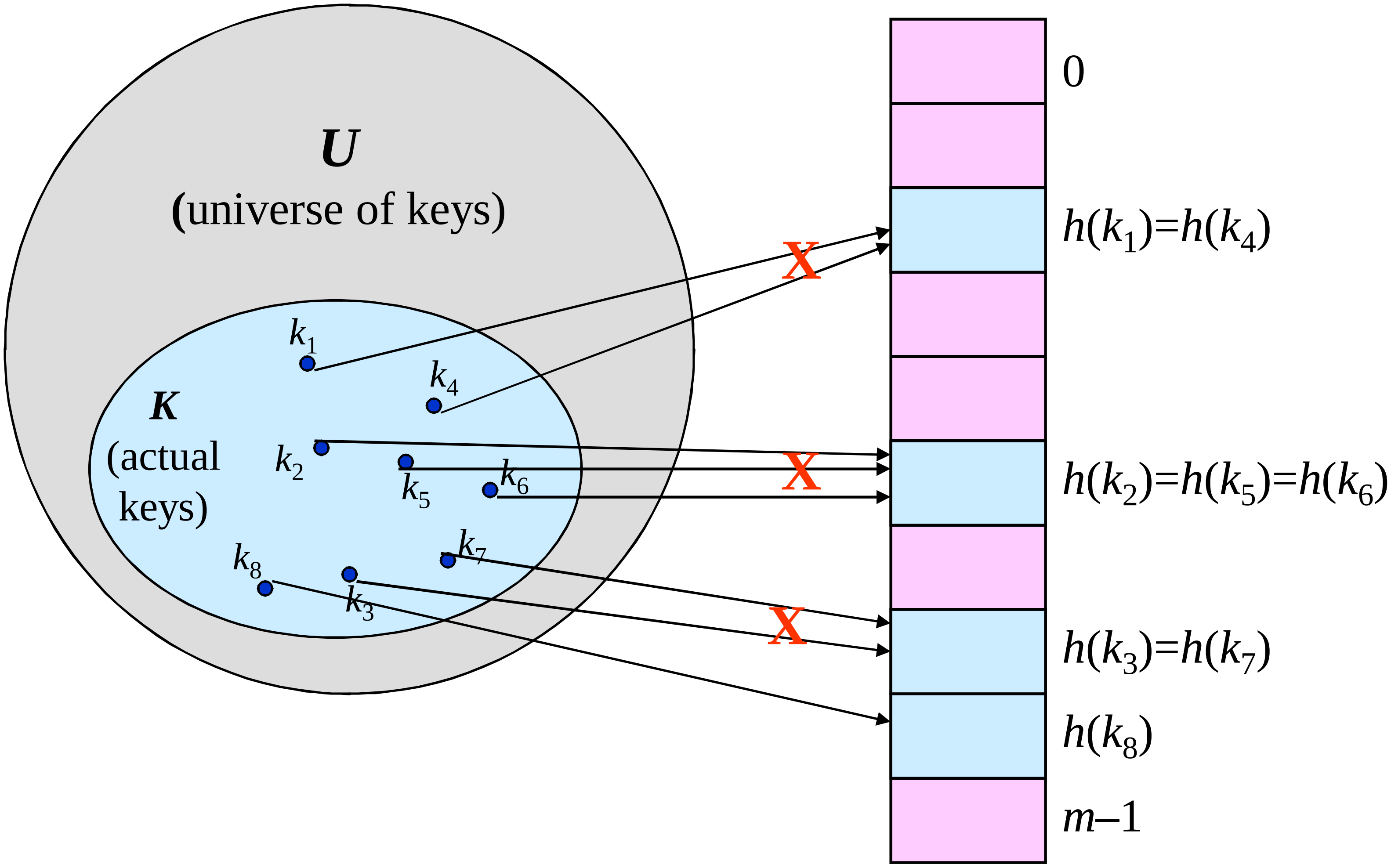
- Open Addressing:
  - All elements are stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



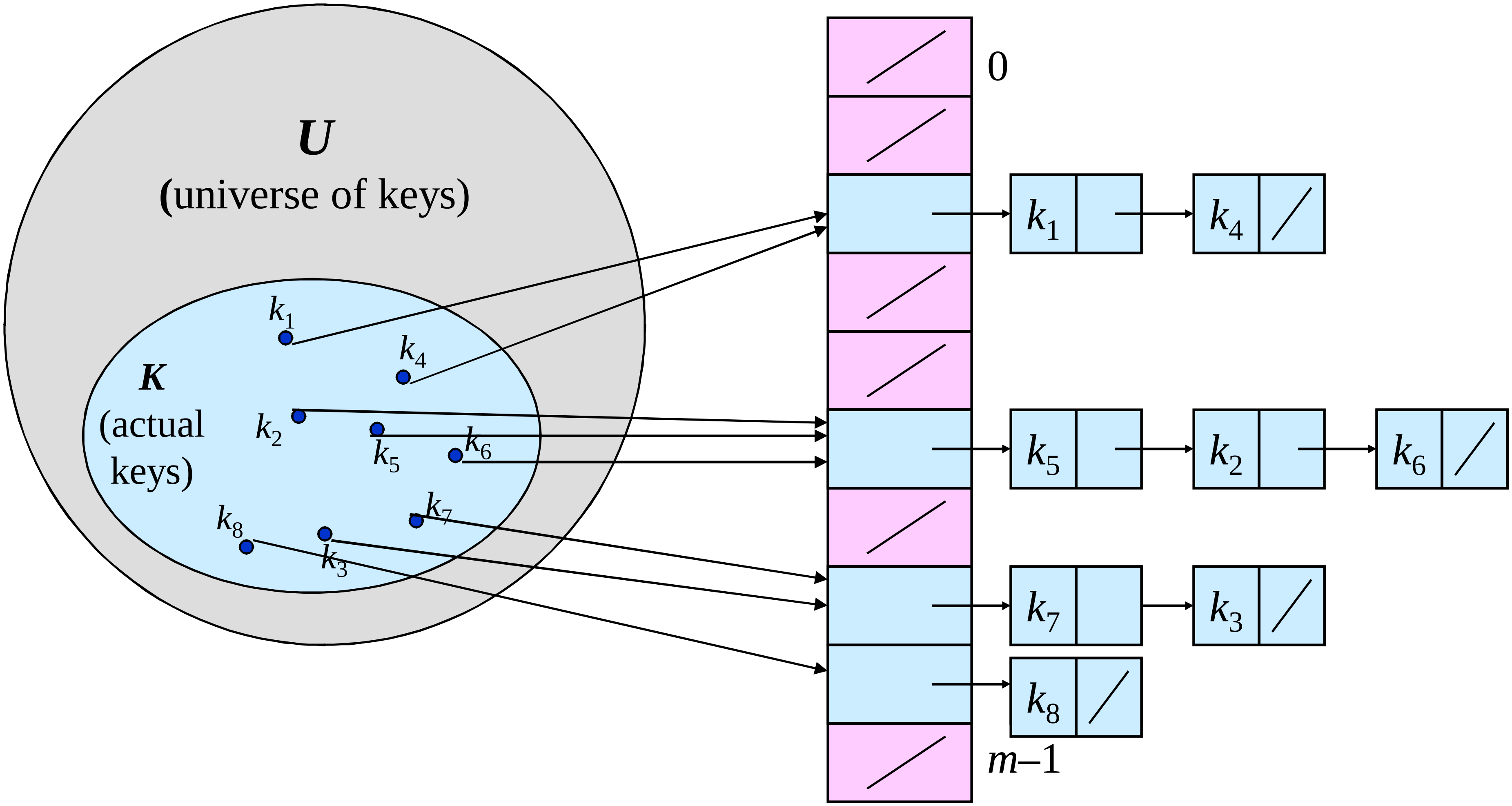
Open Addressing

- Basic idea:
  - To insert: if slot is full, try another slot, …, until an open slot is found (*probing*)
  - To search, follow same sequence of probes as would be used when inserting the element
    - If reach element with correct key, return it
    - If reach a NULL pointer, element is not in table
- Good for fixed sets (adding but no deletion)
  - Example: spell checking
- Table needn't be much bigger than *n*

Collision Resolution by Chaining

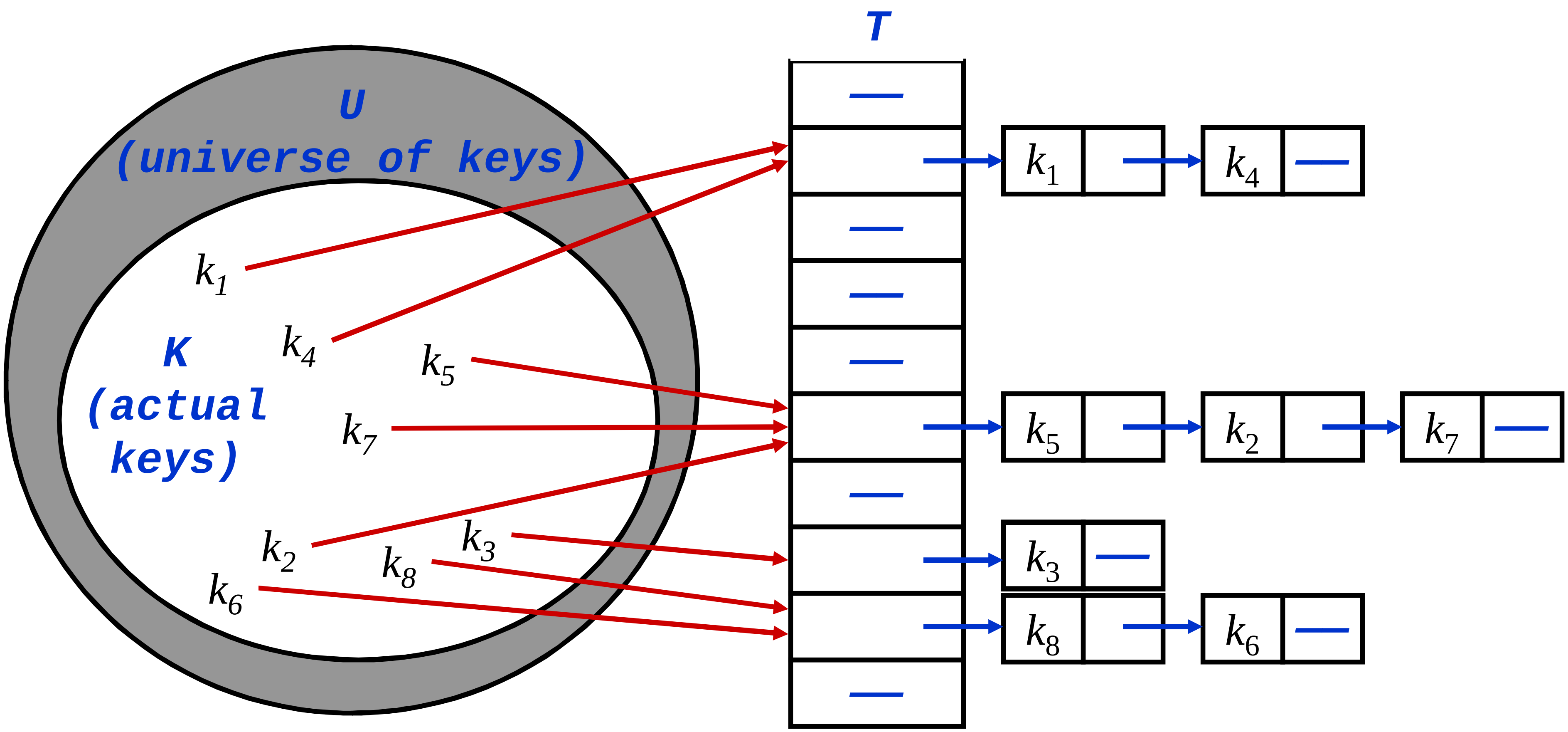


Collision Resolution by Chaining



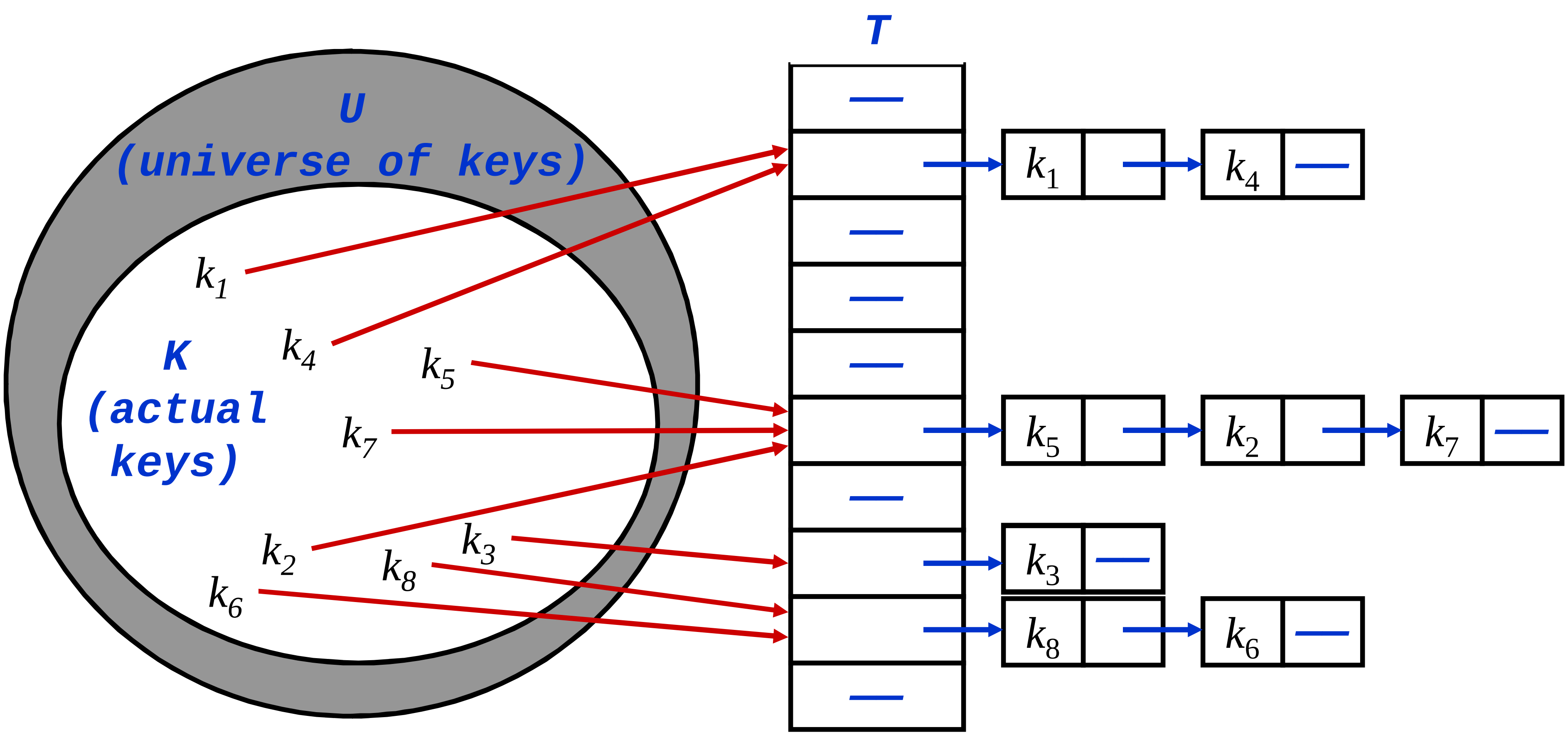
Chaining

- Chaining puts elements that hash to the same slot in a linked list:



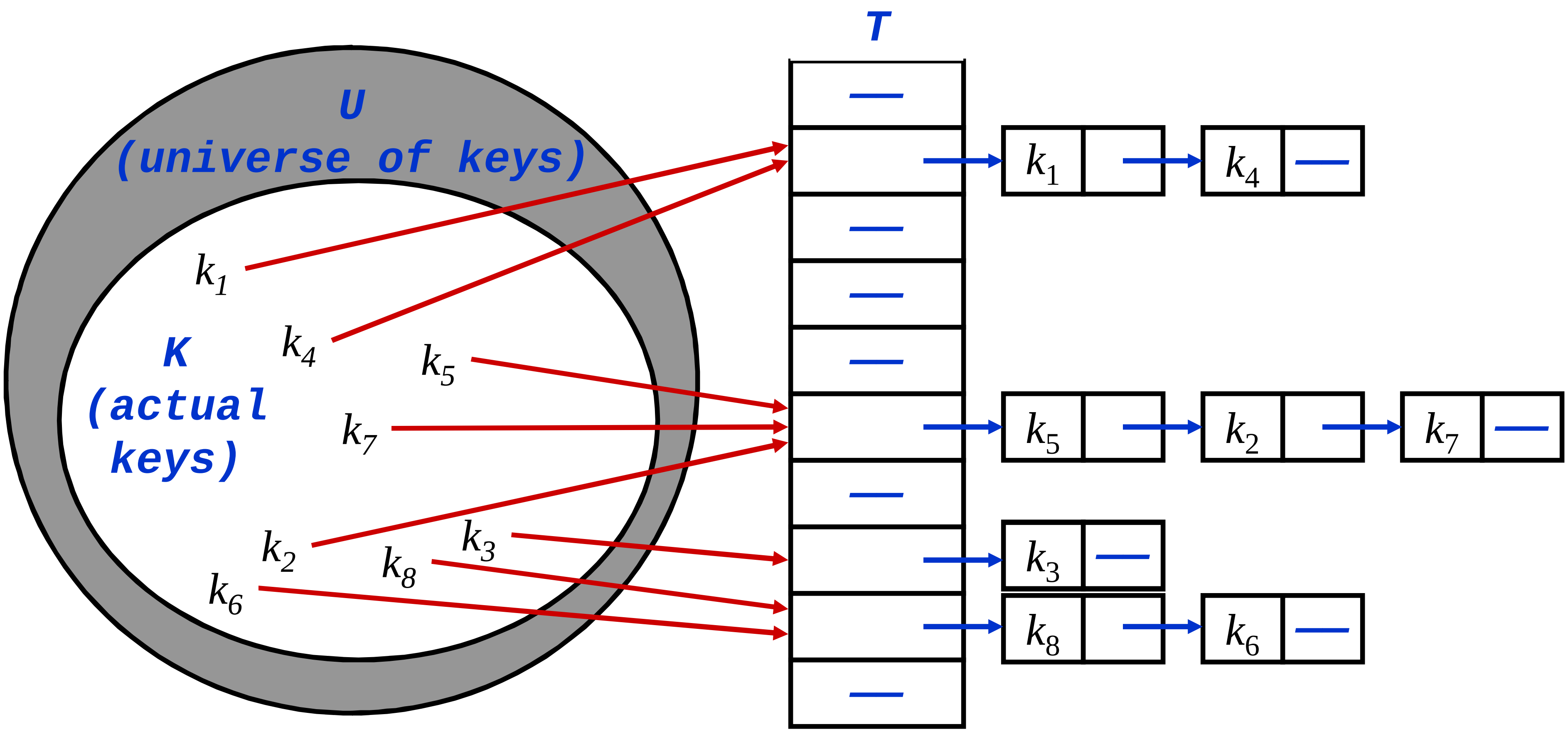
Chaining

- *How do we insert an element?*



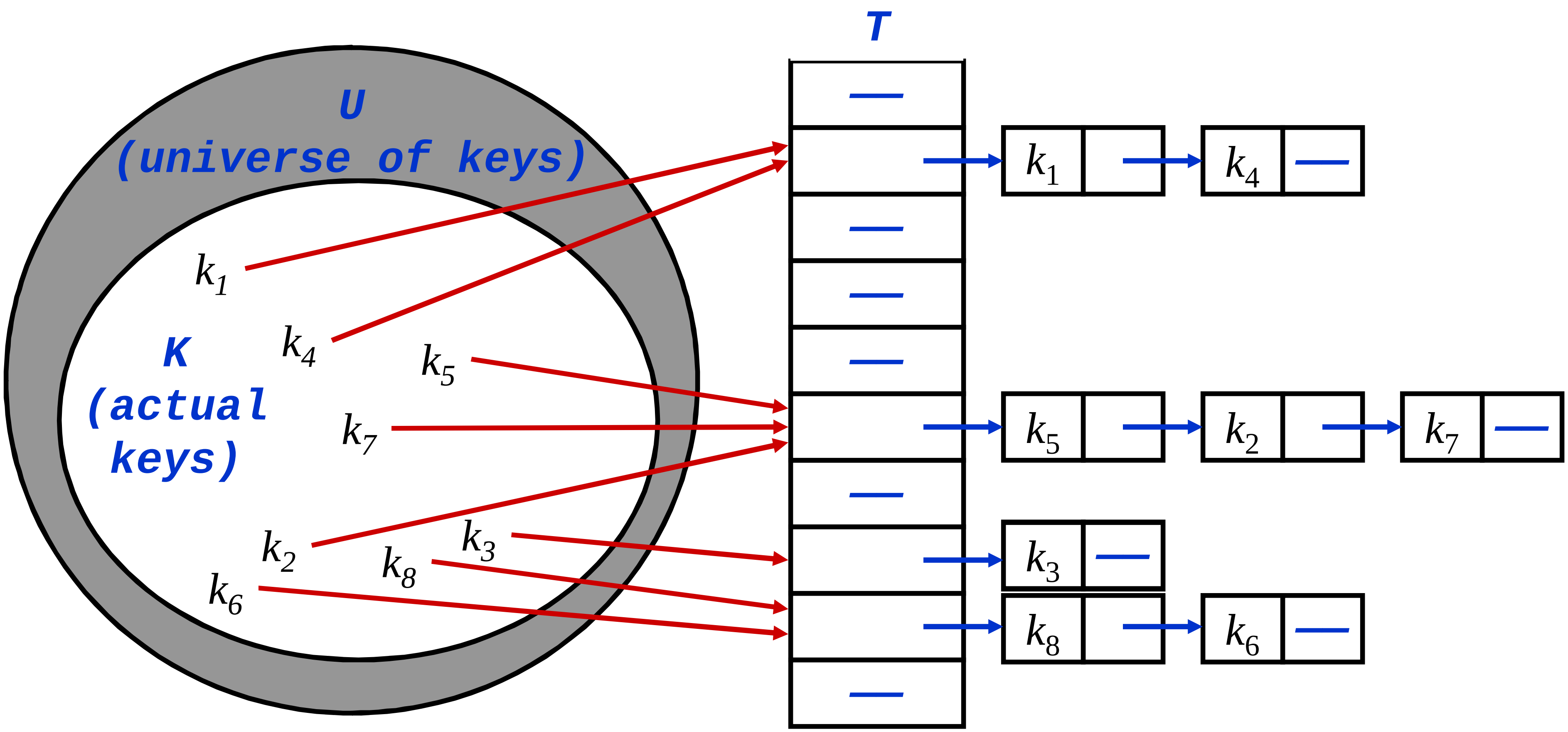
Chaining

- *How do we delete an element?*



Chaining

- *How do we search for a element with a given key?*



Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table: the *load factor* $\alpha = n/m$ = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?*

Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table, the *load factor* $\alpha = n/m$ = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?* A: $O(1+\alpha)$

Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table, the *load factor* $\alpha = n/m$ = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?* A: $O(1+\alpha)$
- *What will be the average cost of a successful search?*

Analysis of Chaining

- Assume *simple uniform hashing*: each key in table is equally likely to be hashed to any slot
- Given n keys and m slots in the table, the *load factor* $\alpha = n/m$ = average # keys per slot
- *What will be the average cost of an unsuccessful search for a key?* A: $O(1+\alpha)$
- *What will be the average cost of a successful search?* A: $O(1 + \alpha/2) = O(1 + \alpha)$

Analysis of Chaining

Draw the 11-item hash table that results from using the hash function $h(i) = (2i + 5) \bmod 11$, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.