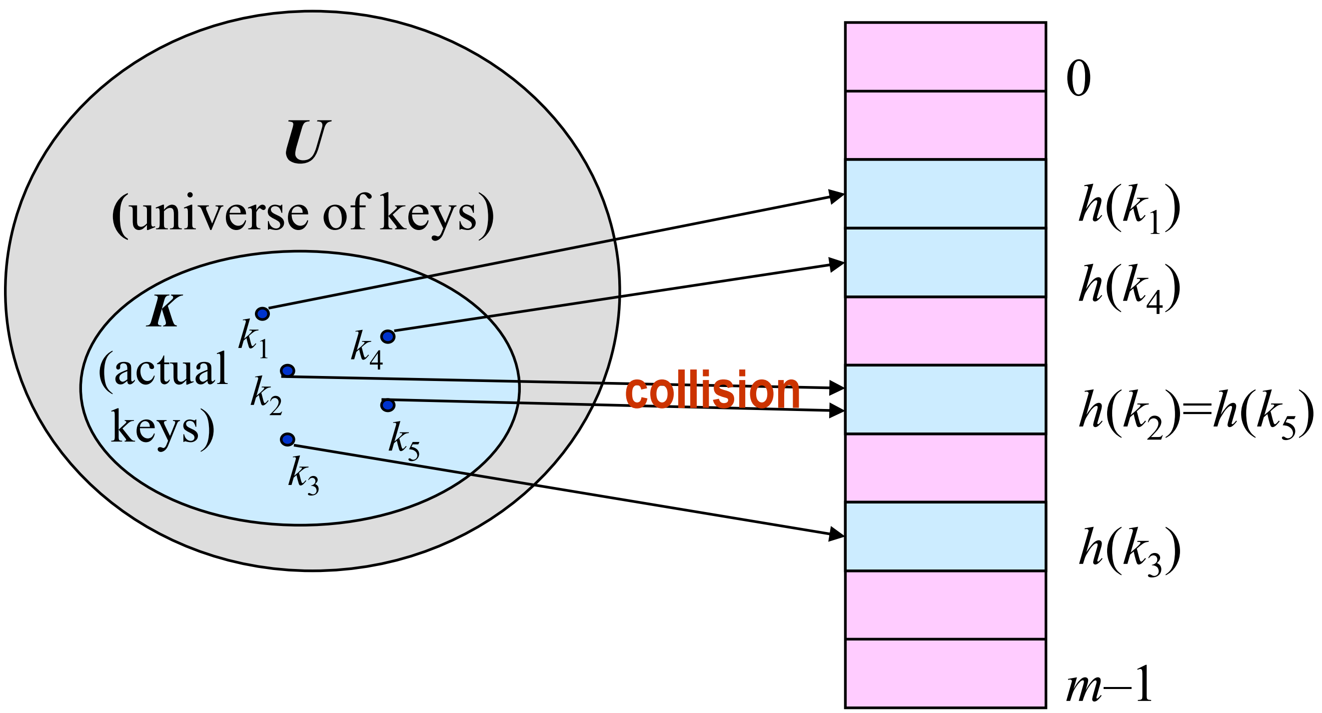
# Hash Tables



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

| | ArrayList | LinkedList |
|---|---|---|
| get() | O(1) | O(n) |
| add() | O(1) | O(1) amortized |
| remove() | O(n) | O(n) |

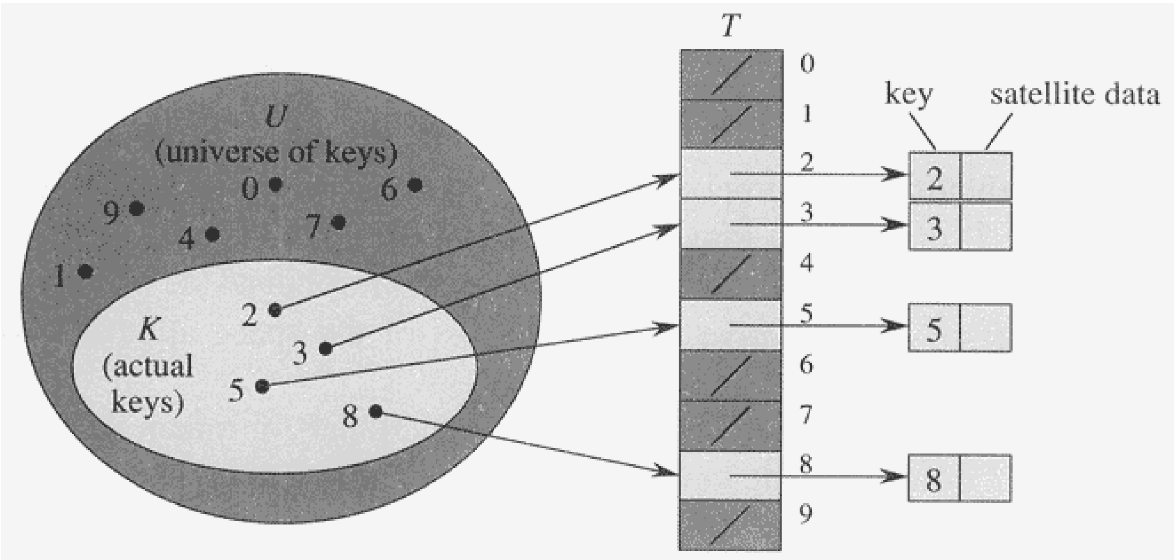
# Dictionary

- **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*

# 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.

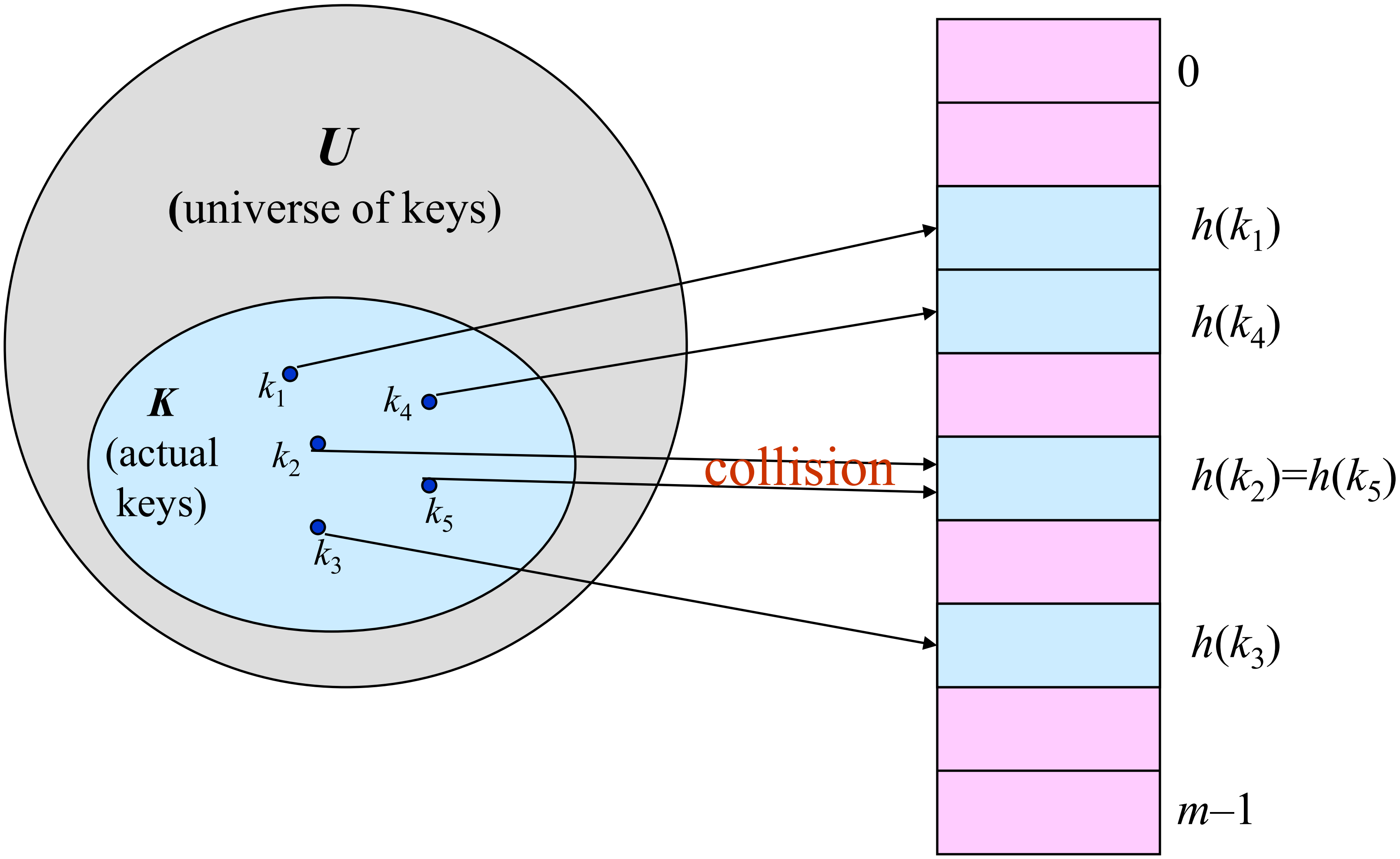
# Hashing

- 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
$h(k_1)$
$h(k_4)$
$h(k_2)=h(k_5)$
$h(k_3)$
$m-1$

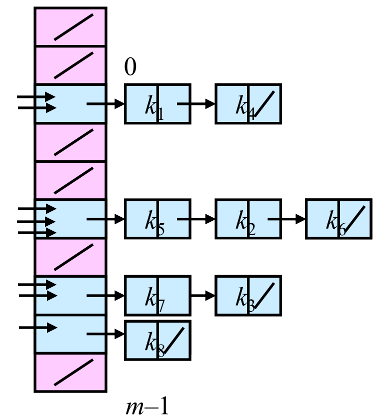
# Issues with Hashing

- 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.

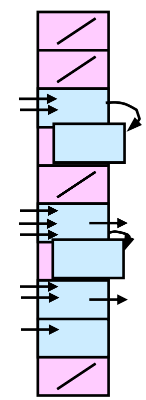
- Search will cost $\Theta(n)$ time in the worst case.
  - However, all operations can be made to have an expected complexity of $\Theta(1)$.

# 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.

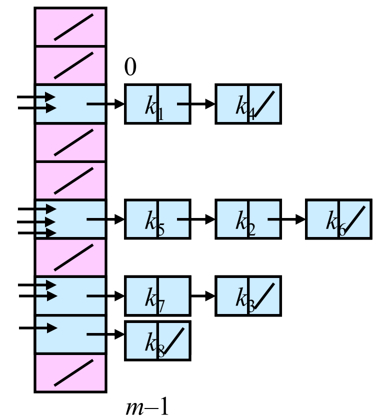



- Open Addressing:
  - All elements are stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.

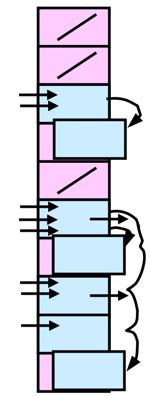


# Methods of Resolution

- 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.

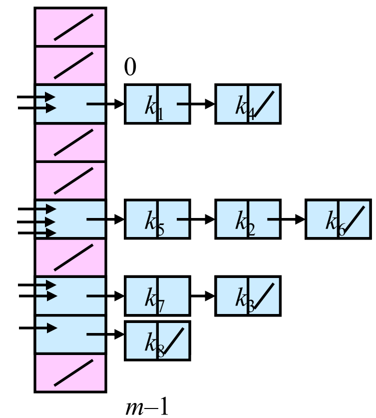



- Open Addressing:
  - All elements are stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.

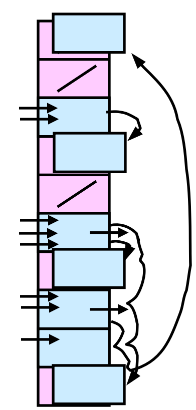


# Methods of Resolution

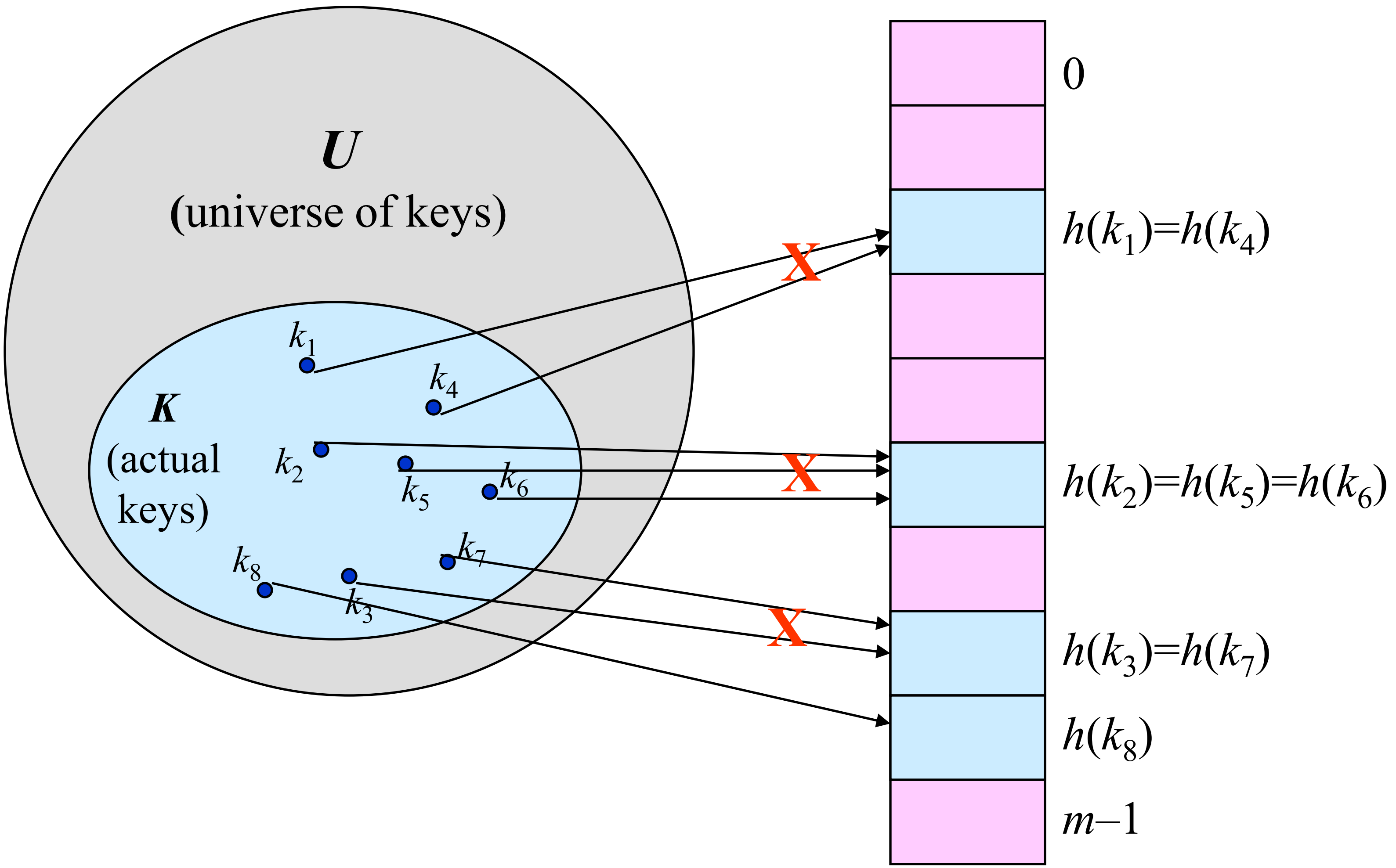
- Chaining:
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.




- 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.

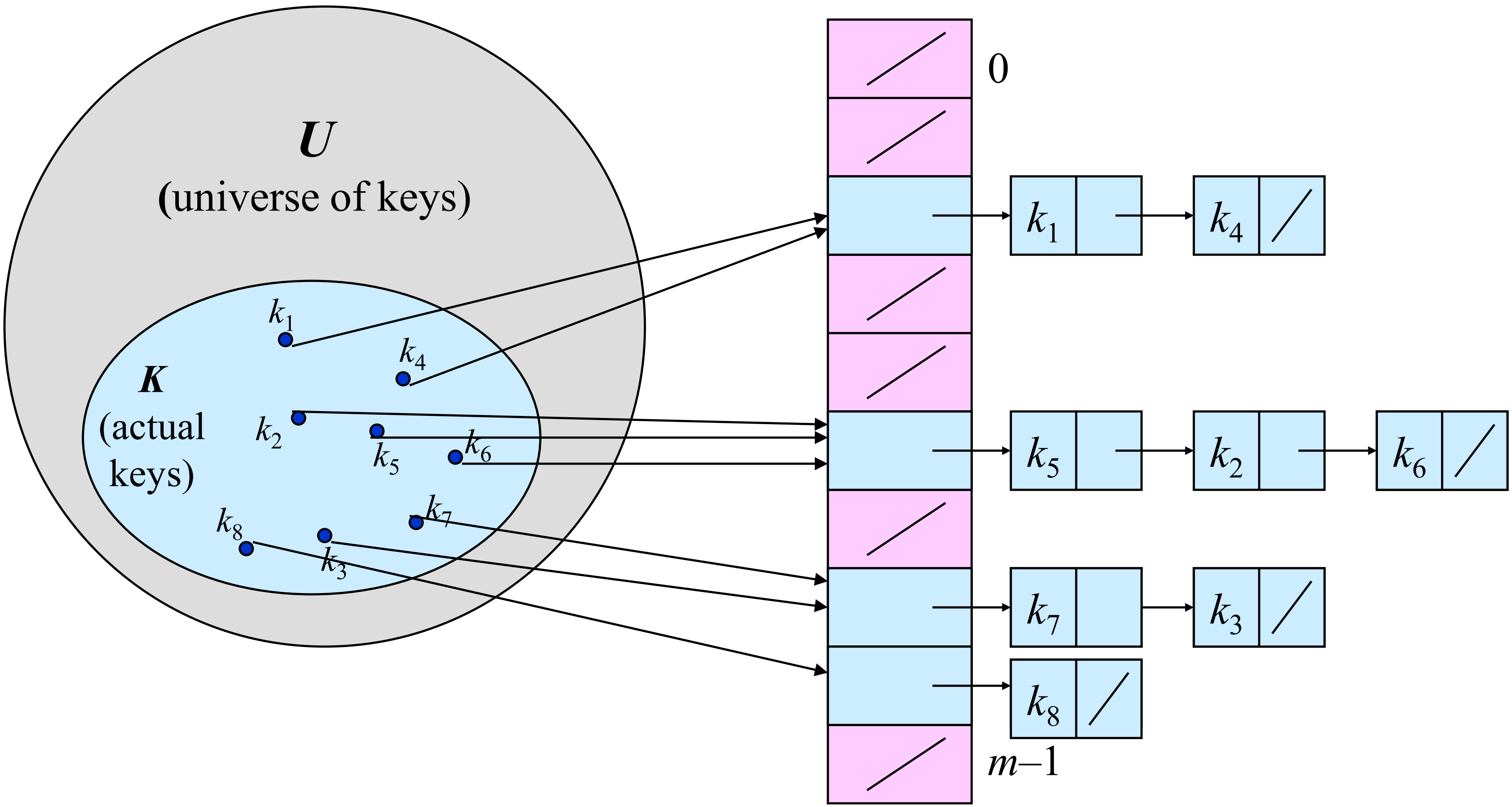


# Collision Resolution by Chaining



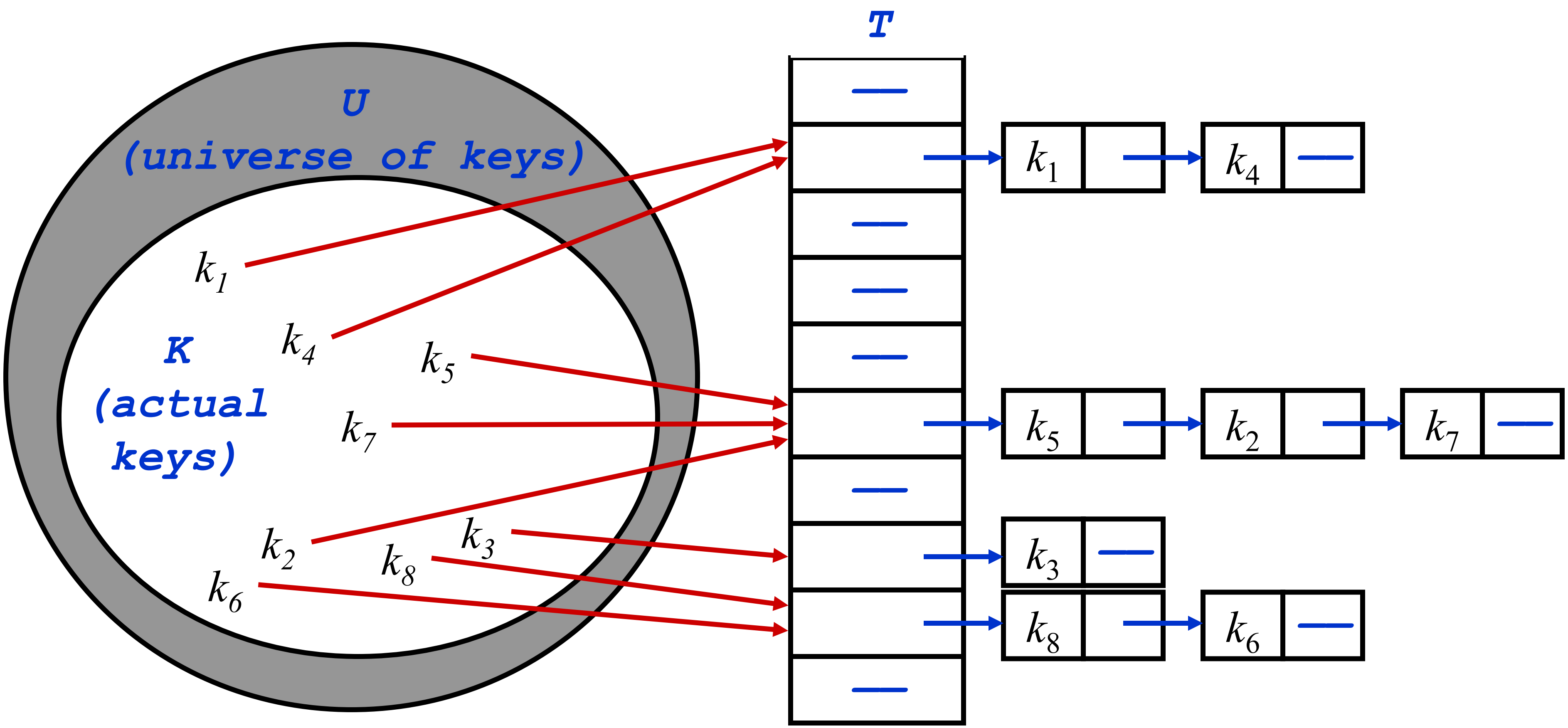
# Collision Resolution by Chaining

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



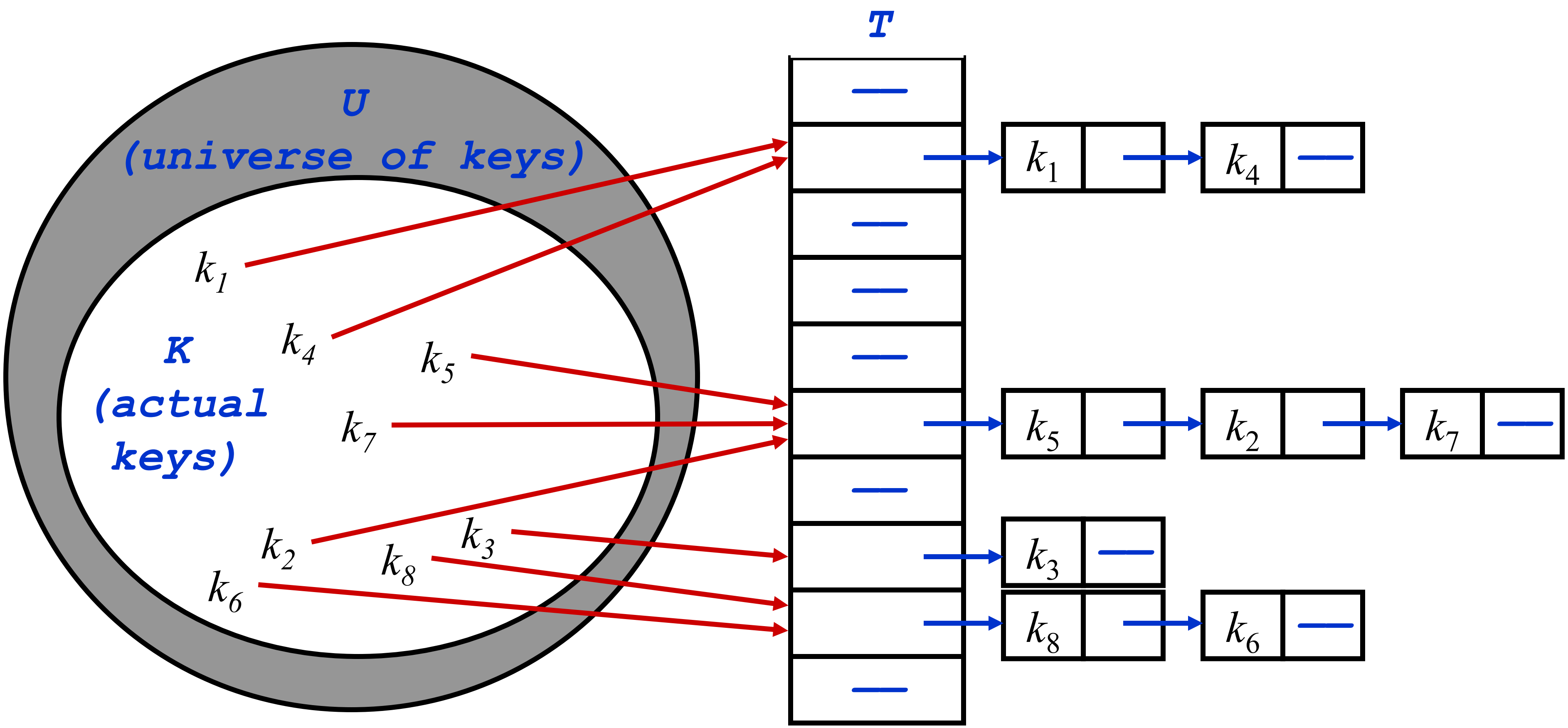
# Chaining

- *How do we insert an element?*



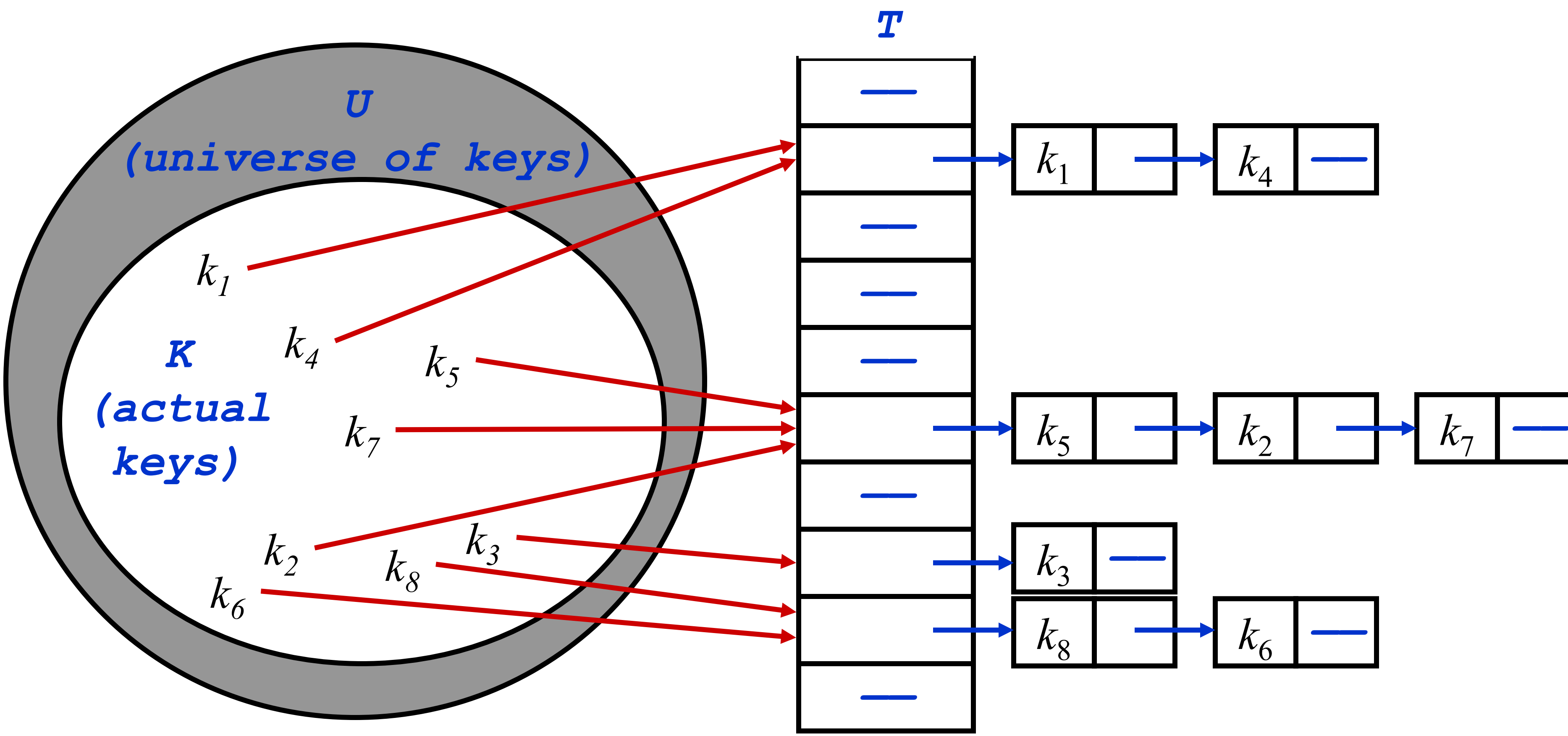
# Chaining

- *How do we delete an element?*



# Chaining

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



# Chaining Example

- $h(k) = 2k\%5$
- keys:
  - 2, 15, 23, 40, 62, 75

# 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 $h(k) = (2k + 5) \bmod 11$, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.

# Analysis of Chaining

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

# 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 $n$

# Probe Sequence

- Sequence of slots examined during a key search constitutes a ***probe sequence***.
- Probe sequence must be a permutation of the slot numbers.
  - We examine every slot in the table, if we have to.
  - We don't examine any slot more than once.

- The hash function is extended to:
  - $\boldsymbol{h : U \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{probe number}} \rightarrow \underbrace{\{0, 1, \ldots, m-1\}}_{\text{slot number}}}$

- $\langle h(k, 0), h(k, 1), \ldots, h(k, m-1) \rangle$ should be a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.

# Computing Probe Sequences

- The ideal situation is ***uniform hashing***:
  - Generalization of simple uniform hashing.
  - Each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$ as its probe sequence.
- It is hard to implement true uniform hashing.
  - Approximate with techniques that at least guarantee that the probe sequence is a permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- Some techniques:
  - Use ***auxiliary hash functions***.
    - Linear Probing.
    - Quadratic Probing.
    - Double Hashing.
  - Can't produce all $m!$ probe sequences.

# Linear Probing

- $h(k, i) = (h'(k) + i) \bmod m.$

  key Probe number Auxiliary hash function

- The initial probe determines the entire probe sequence.
  - $T[h'(k)], T[h'(k)+1], \ldots, T[m-1], T[0], T[1], \ldots, T[h'(k)-1]$
  - Hence, only $m$ distinct probe sequences are possible.
- Suffers from ***primary clustering***:
  - Long runs of occupied sequences build up.
  - Long runs tend to get longer, since an empty slot preceded by $i$ full slots gets filled next with probability $(i+1)/m$.
  - Hence, average search and insertion times increase.

# Ex: Linear Probing

- Example:
  - $\boldsymbol{h'}(\boldsymbol{k}) = \boldsymbol{k} \bmod 13$
  - $h(k, i) = (\boldsymbol{h'}(\boldsymbol{k}) + i) \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

$\Downarrow$

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 41 | | | 18 | 44 | 59 | 32 | 22 | 31 | 73 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# Operation Insert

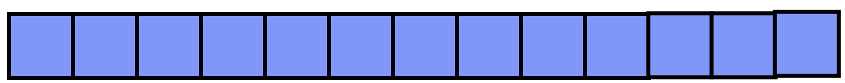
- Act as though we were searching, and insert at the first NIL slot found.

Hash-Insert($T$, $k$)
1. $i \leftarrow 0$
2. **repeat** $j \leftarrow h(k, i)$
3.     **if** $T[j]$ = NIL
4.         **then** $T[j] \leftarrow k$
5.             **return** $j$
6.         **else** $i \leftarrow i + 1$
7. **until** $i = m$
8. **error** "hash table overflow"

# Pseudo-code for Search

```
Hash-Search (T, k)
1. i ← 0
2. repeat j ← h(k, i)
3.           if T[j] = k
4.              then return j
5.           i ← i + 1
6. until T[j] = NIL or i = m
7. return NIL
```

- Example:
  - $\boldsymbol{h'}(\boldsymbol{k}) = \boldsymbol{k} \text{ mod } 13$
  - $h(k, i) = (\boldsymbol{h'}(\boldsymbol{k}) + i) \text{ mod } 13$
  - Insert keys 17,30,43 in this order; delete 30; search 43;



# Deletion

- Cannot just turn the slot containing the key we want to delete to contain NIL. Why?
- Use a special value DELETED instead of NIL when marking a slot as empty during deletion.
  - ***Search*** should treat DELETED as though the slot holds a key that does not match the one being searched for.
  - ***Insert*** should treat DELETED as though the slot were empty, so that it can be reused.
- **Disadvantage:** Search time is no longer dependent on $\alpha$.
  - Hence, chaining is more common when keys have to be deleted.

# Quadratic Probing

- $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \quad c_1 \neq c_2$

key Probe number Auxiliary hash function

- The initial probe position is $T[h'(k)]$, later probe positions are offset by amounts that depend on a quadratic function of the probe number $i$.
- Must constrain $c_1$, $c_2$, and $m$ to ensure that we get a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
- Can suffer from ***secondary clustering***:
  - If two keys have the same initial probe position, then their probe sequences are the same.

- $h(k, i) = (h'(k) + i^2) \bmod 7$
- $h'(k) = k\%7$
- *insert keys: 76, 40, 48, 5, 55*

| **0** | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| 48 | | 5 | 55 | | 40 | 76 |

- *insert keys: 76, 40, 48, delete key 76 (replace with NIL), search 48*
  - *not found*

| **0** | **1** | **2** | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|---|
| 48 | | | | | 40 | ~~76~~NIL |

# Double Hashing

- $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$

  key Probe number Auxiliary hash functions
- Two auxiliary hash functions.
  - $h_1$ gives the initial probe. $h_2$ gives the remaining probes.
- Must have $h_2(k)$ relatively prime to $m$, so that the probe sequence is a full permutation of $\langle 0, 1, \ldots, m-1 \rangle$.
  - Choose $m$ to be a power of 2 and have $h_2(k)$ always return an odd number. Or,
  - Let $m$ be prime, and have $1 < h_2(k) < m$.
- $\Theta(m^2)$ different probe sequences.
  - One for each possible combination of $h_1(k)$ and $h_2(k)$.
  - Close to the ideal uniform hashing.

One good choice is to choose a prime R < size and:

$\text{hash}_2(x) = R - (x \bmod R)$

- $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod 7$
- $h_1(k) = k\%7,$
- $h_2(k) = 5 - (k\%5)$
- *keys: 76, 40, 47, 55, 10, 93,*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | 47 | 93 | 10 | 55 | 40 | 76 |

- Consider an open-addressing hash table as shown below. The table already contains four data items. Assume that collisions are handled by the hash function
  $\boldsymbol{h(k,i) = \left(h'(k) + ih2(k)\right) mod\ 13}$, where $\boldsymbol{h'(k) = (2k + 7)\ mod\ 13}$ and $\boldsymbol{h2(k) = (k + 5)\ mod\ 13}$.
  *By showing calculations, redraw the table* after
  - (i) insert 90;
  - (ii) insert 83
- What is collision?
- Chaining vs Open Addressing; pros cons