# Hash Tables

CLRS 11.1, 11.2, 11.4
(+ some supplemental material)


**Hash table**: an **unordered** dictionary which stores a searchable collection of key-element items, implemented via

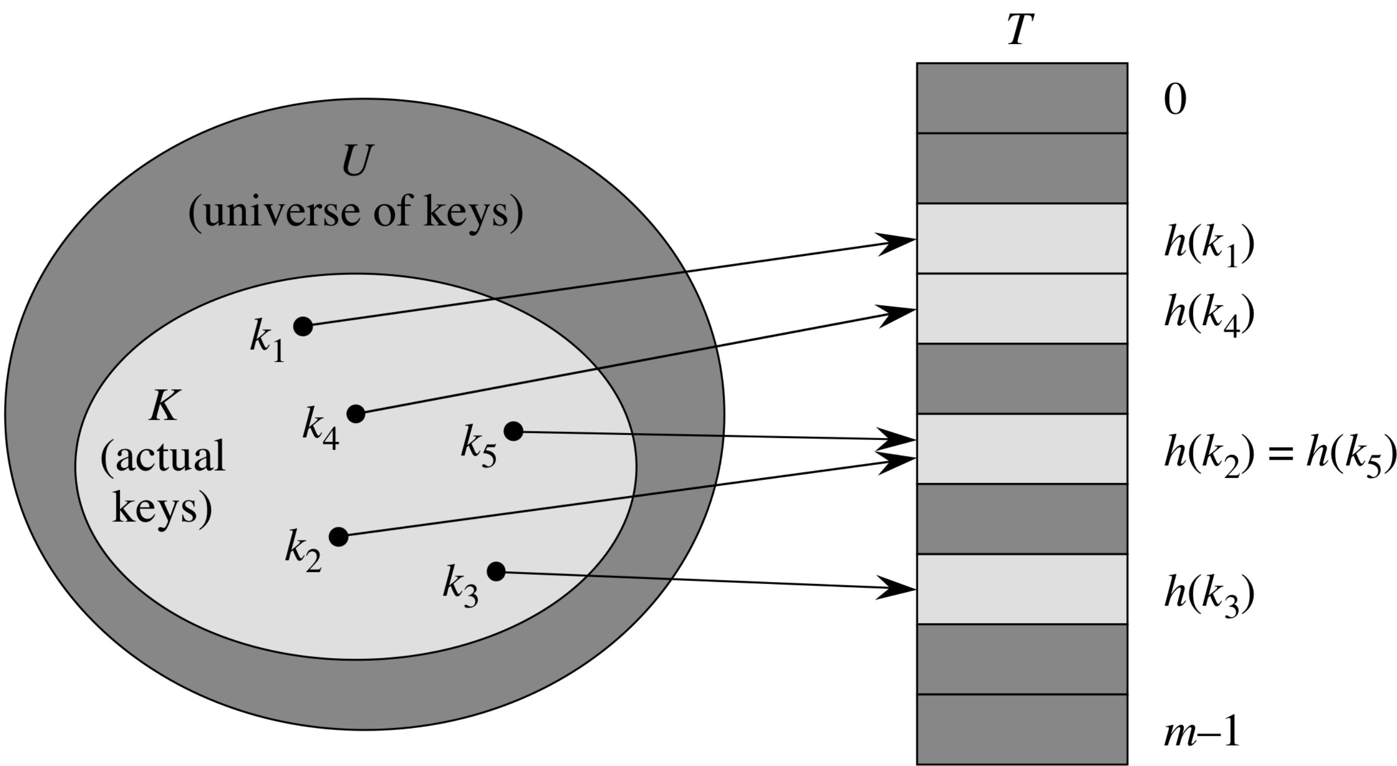
- an *array*, and
- *hash function*.

# Hash Table & Hash Functions

A **hash table** consists of:

- **array** (called table) **T** of size $m$
- **hash function** $h : U \to \{0,1, \ldots , m-1\}$, which maps keys of a given type to integers in a fixed integer interval
  - Ex: $h(x) = x \bmod m$ is a hash function for integer keys
  - Ex: A mapping of all state names to integers 0-49
  - The integer $h(x)$ is called the **hash value** of key *x.* We also say x **hashes** to h(x)

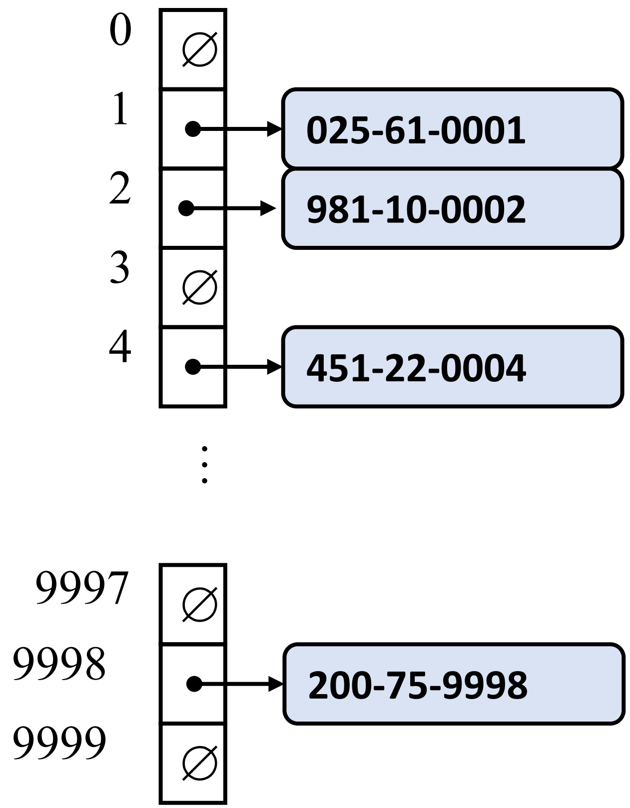
**Goal:**
Store item *(k, o)* at index *i = h(k)* in the table.



# Example hash table

A hash table to store personnel records, where each key *k* is the social security number of the employee.

- Use array of size m=10,000
- Hash function *h(x)* = last four digits of *x*



# The problem of collisions

- A **collision** occurs when two different keys hash to the same slot.

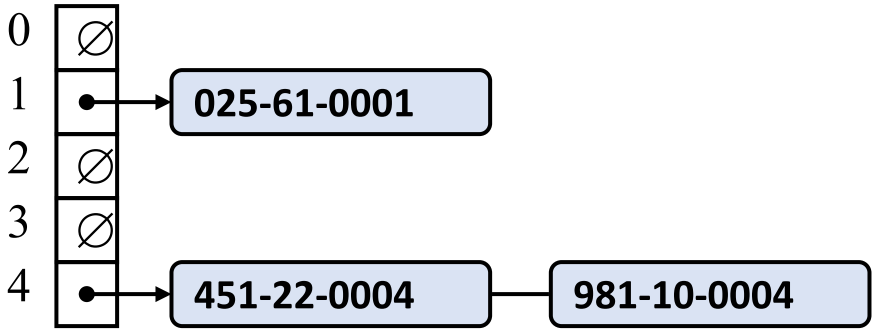
- Prevent collisions → Depends on the hash function
  - A universal hash function reduces the probability of collisions [CLRS 11.3]
  - A perfect hash function guarantees no collisions, at the cost of more memory [CLRS 11.5]

- Handle collisions systematically
  - **Chaining**
    - Each slot may contain multiple items
    - If a collision occurs, append it to the bucket
  - **Open Addressing** (linear probing, quadratic probing, double hashing)
    - Each slot contains at most one item
    - If a collision occurs, find a different slot which is empty
    - Various approaches to finding an empty slot

# Collision Handling

## Chaining

- each cell in the table points to a linked list of elements that map there
- simple, but requires additional memory outside the table




## Open Addressing

- the colliding item is placed in a different cell of the table
- no additional memory, but complicates searching/removing
- common types: linear probing, quadratic probing, double hashing

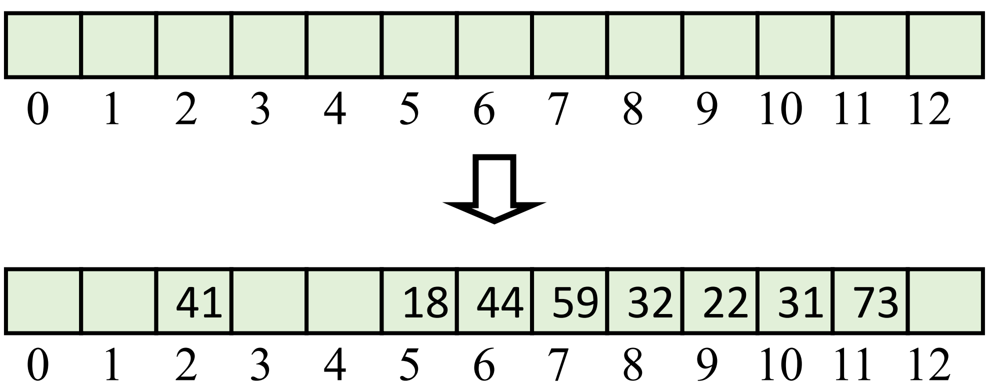
# Open addressing: linear probing

- Place the colliding item in the next (circularly) available table cell

  try $T[\,(h(k)+i) \bmod m]$ for $i = 0,1,2,\ldots$

- Colliding items cluster together, causing future collisions to cause a longer sequence of probes (searches for next available cell)

- Example:
  - *h(x) = x mod 13*
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order




*h(18)* = 18 mod 13 = 5
41 mod 13 = 2
22 mod 13 = 9
44 mod 13 = 5
59 mod 13 = 7
32 mod 13 = 6
31 mod 13 = 5
73 mod 13 = 8

# Searching for an item

- Start at cell *h(k)*
- Check consecutive locations until one of the following occurs
  - An item with key *k* is found, or
  - An empty cell is found, or
  - *m* cells have been unsuccessfully probed

# Open addressing: double hashing

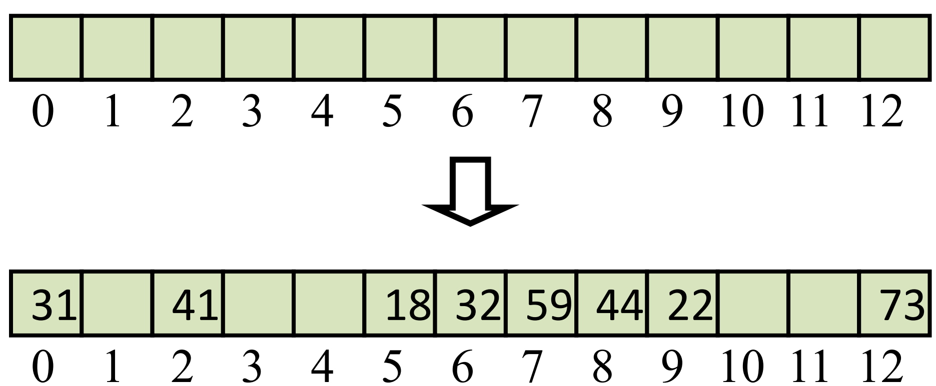
- Use a secondary hash function $d(k)$ to place items in first available cell

  try $T\left[\left(h(k) + i \cdot d(k)\right) \bmod m\right]$ for $i = 0,1,2,\ldots$

- $d(k)$ cannot have zero values
- The table size $m$ must be a prime to allow probing of all the cells

- Example:
  - $h(k) = k \bmod 13$
  - $d(k) = 1 + (k \bmod 7)$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

| $k$ | $h(k)$ | $d(k)$ | Probes | | |
|---|---|---|---|---|---|
| 18 | 5 | 5 | 5 | | |
| 41 | 2 | 7 | 2 | | |
| 22 | 9 | 2 | 9 | | |
| 44 | 5 | 3 | 5 | 8 | |
| 59 | 7 | 4 | 7 | | |
| 32 | 6 | 5 | 6 | | |
| 31 | 5 | 4 | 5 | 9 | 0 |
| 73 | 8 | 4 | 8 | 12 | |



# Performance of hashing

- In the worst case, searches, insertions and removals on a hash table take O(n) time
  - occurs when all inserted keys collide
- The **load factor** $\alpha = n/m$ affects the performance of a hash table
  - Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is $\frac{1}{1-\alpha}$
  - The expected number of probes for an insertion with chaining is $O(1+\alpha)$
- The **expected running time** of all the dictionary ADT operations in a hash table is **O(1)**
- In practice, hashing is very fast provided the load factor is not close to 100%

# Other

How efficiently can you solve these common interview questions?

*Hint: I selected these ones because there is an approach which uses a hash table*

- You are given an array A of integers. Determine the integer that occurs most frequently in A.
- You are given an array A of integers, and a number $x$. Determine whether there exists two elements in A whose sum is exactly $x$.