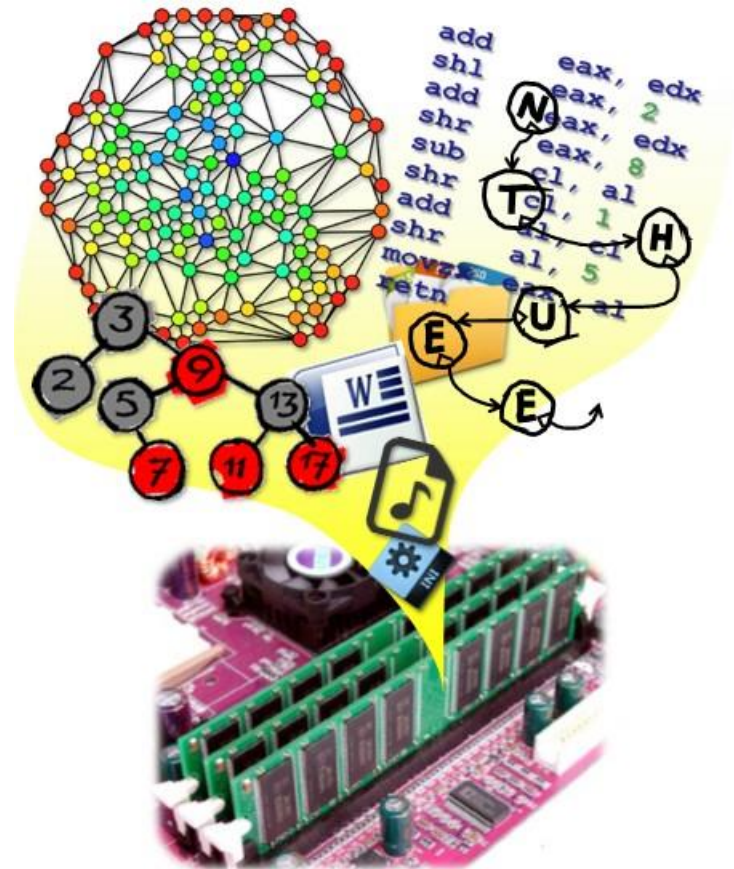# Data Structures

CH8 Hashing

Prof. Ren-Shuo Liu

NTHU EE

Spring 2018

Outline

- 8.1 Introduction
- (8.2 Static hashing)
- (8.3 Dynamic hashing)
- 8.4 Bloom filters

Registration Division Example

請大家向註冊組承辦人
查詢學期成績

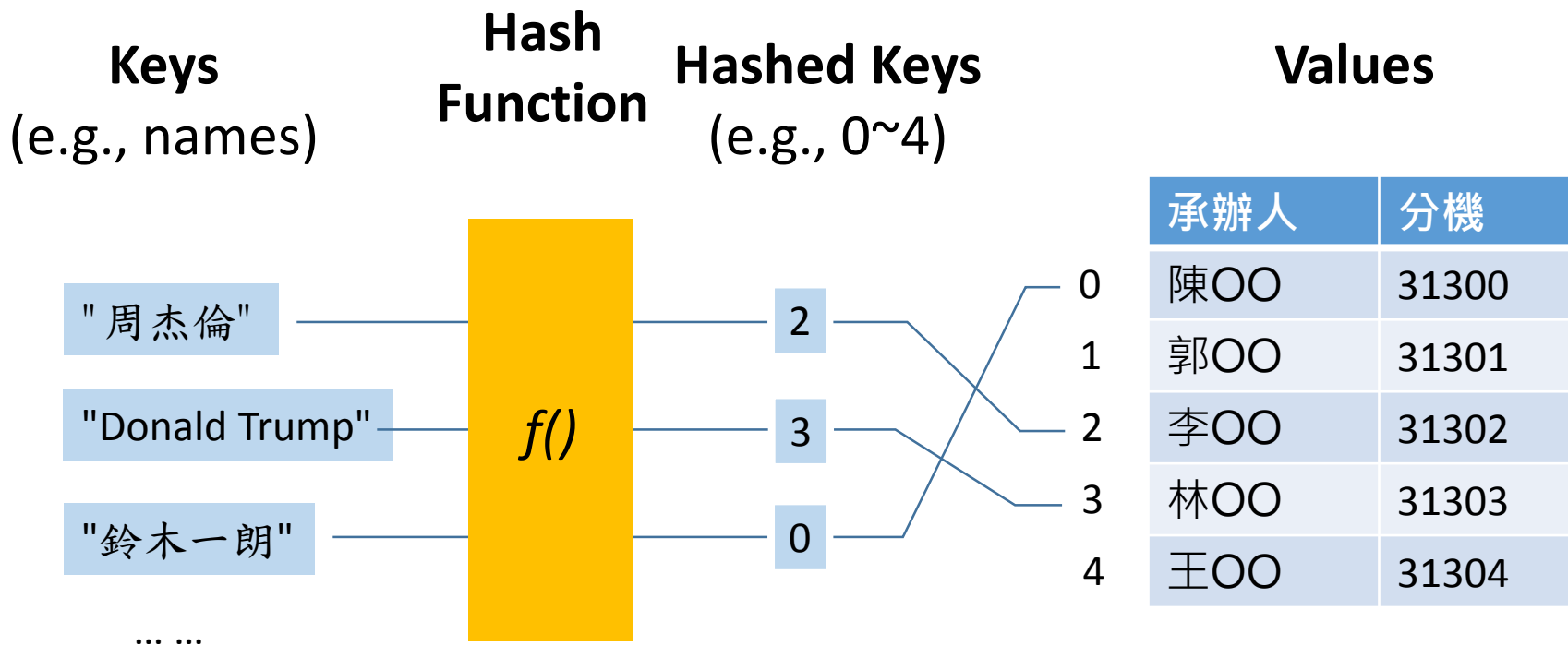| 承辦人	分機 / Email
陳OO	31300 / chen@nthu...
郭OO	31301 / kuo@nthu...
李OO	31302 / li@nthu...
林OO	31303 / lin@nthu..
王OO	31304 / wang@nthu...

Registration Division Example

全校多數都打電話、
寄email給第一位

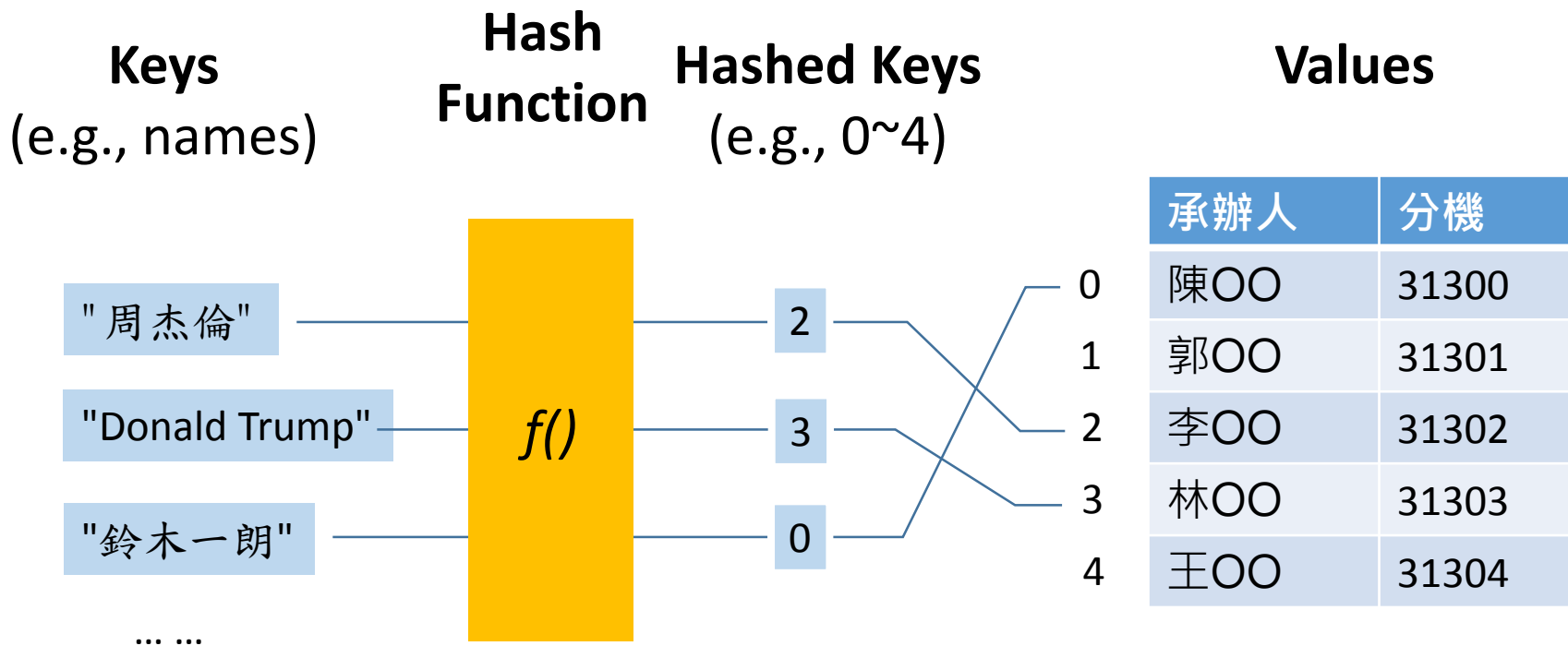| 承辦人	分機 / Email
陳OO	31300 / chen@nthu...
郭OO	31301 / kuo@nthu...
李OO	31302 / li@nthu...
林OO	31303 / lin@nthu..
王OO	31304 / wang@nthu...

Hash Concepts

- **Hash function**
  - Any deterministic function that can map data of arbitrary size (original keys) to data of a desired fixed size (hashed keys)

Keys (e.g., names)	Hash Function	Hashed Keys (e.g., 0~4)	Values

承辦人	分機
陳OO	31300
郭OO	31301
李OO	31302
林OO	31303
王OO	31304

"周杰倫"

"Donald Trump"

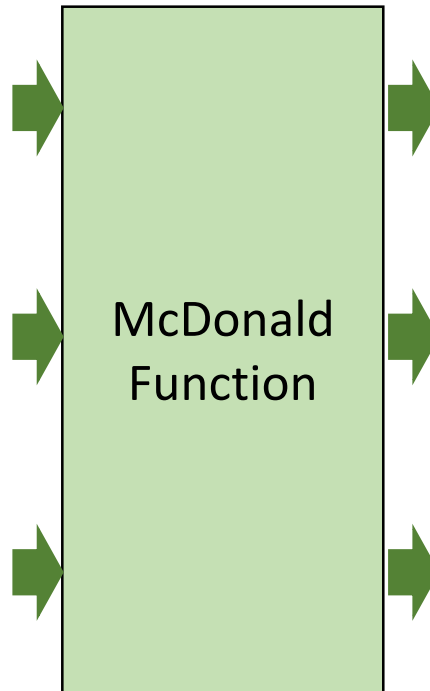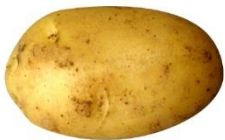"鈴木一朗"

… …

f()

2

3

0

0
1
2
3
4

Hash Concepts

- **Hash function**
  - It shuffles the order of mapping
  - But it is deterministic

Keys (e.g., names)	Hash Function	Hashed Keys (e.g., 0~4)		Values

承辦人	分機
陳OO	31300
郭OO	31301
李OO	31302
林OO	31303
王OO	31304

"周杰倫" → f() → 2

"Donald Trump" → f() → 3

"鈴木一朗" → f() → 0

… …

Hash in Cooking

- Hash: "chop and mix foods"

- Example: hash browns (薯餅)

McDonald Function
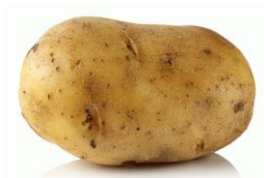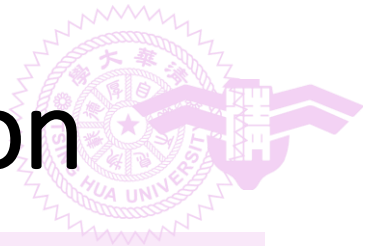
Hash in Chinese Decomposition

- Decompose Chinese characters into keyboard strokes
  - Facilitate Chinese input
- Example: the Boshiamy (嘸蝦米) decomposition scheme
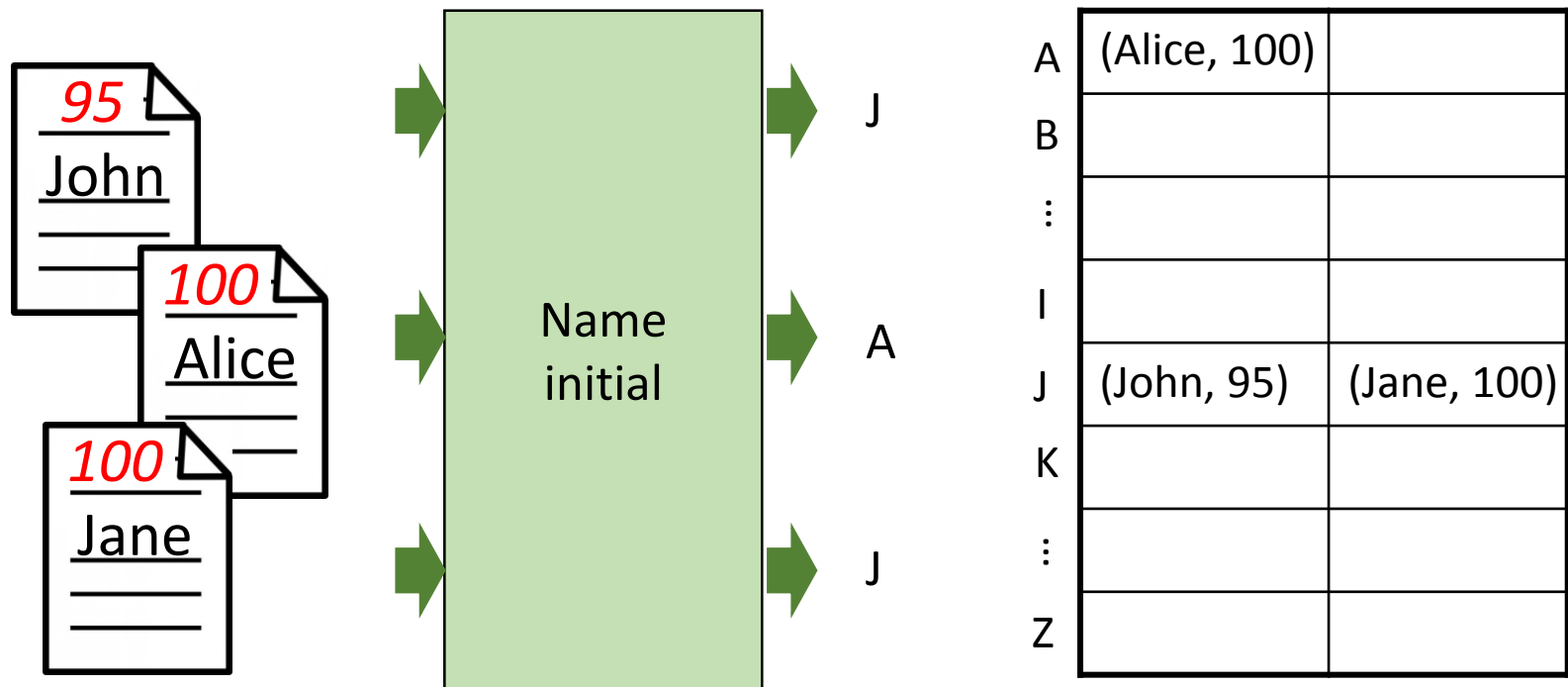
哈 → Boshiamy Function → OAO

州 → Boshiamy Function → YYY

哥 → Boshiamy Function → TOTO
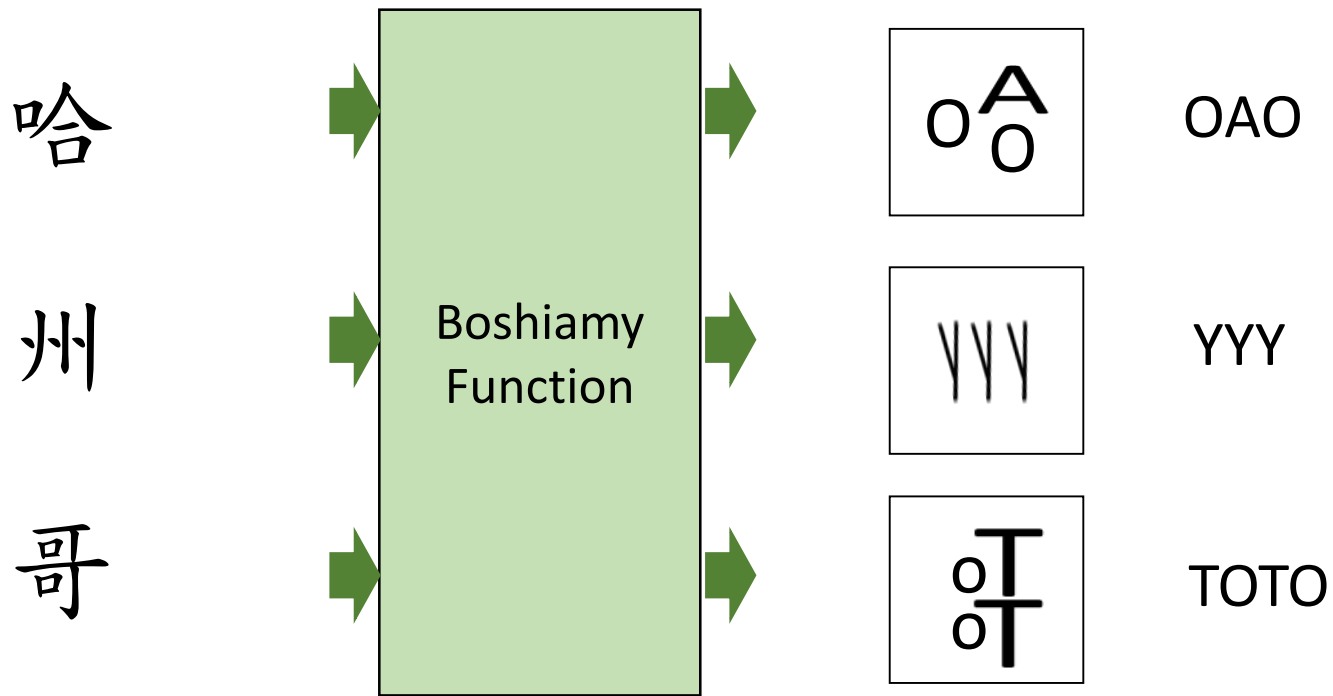
Hash in a Storing Data

- Example: Storing students' grades according to their name initial letters

Advantages of Hashing

- Inserting, deleting, and searching can be ~O(1) time
  - Hash function computation is designed in O(1)
  - Indexing the corresponding bucket in the table is O(1)
  - Searching all slots in a bucket for a key is also O(1)
    - The number of slots is independent of the number of pairs stored in the table

A	(Alice, 100)	
B	(Bob, 80)	(Ben, 70)
⋮		
I	(Irene, 85)	
J	(John, 95)	(Jane, 100)
K	(Ken, 75)	
⋮		
Z	(Zoe, 80)	

Hashing

- A pair with a key k is stored in a hash table *ht*

- Key parameters
  - *b* buckets in *ht*
  - *h(k)* is the home bucket of a key *k*
  - *s* slots per bucket
  - *T* possible different keys
  - *n* stored pairs in *ht*

Slots

A	(Alice, 100)	
B	(Bob, 80)	(Ben, 70)
⋮		
I	(Irene, 85)	
J	(John, 95)	(Jane, 100)
K	(Ken, 75)	
⋮		
Z	(Zoe, 80)	

Buckets

Hash Function

定義域
通常為不限大小

Hash Function
通常必須"多對一"
絕不會"一對多"

值域
通常為有限大小
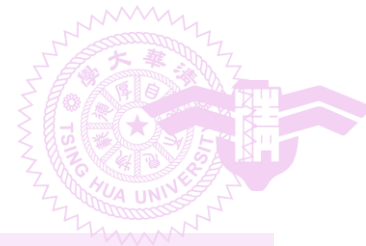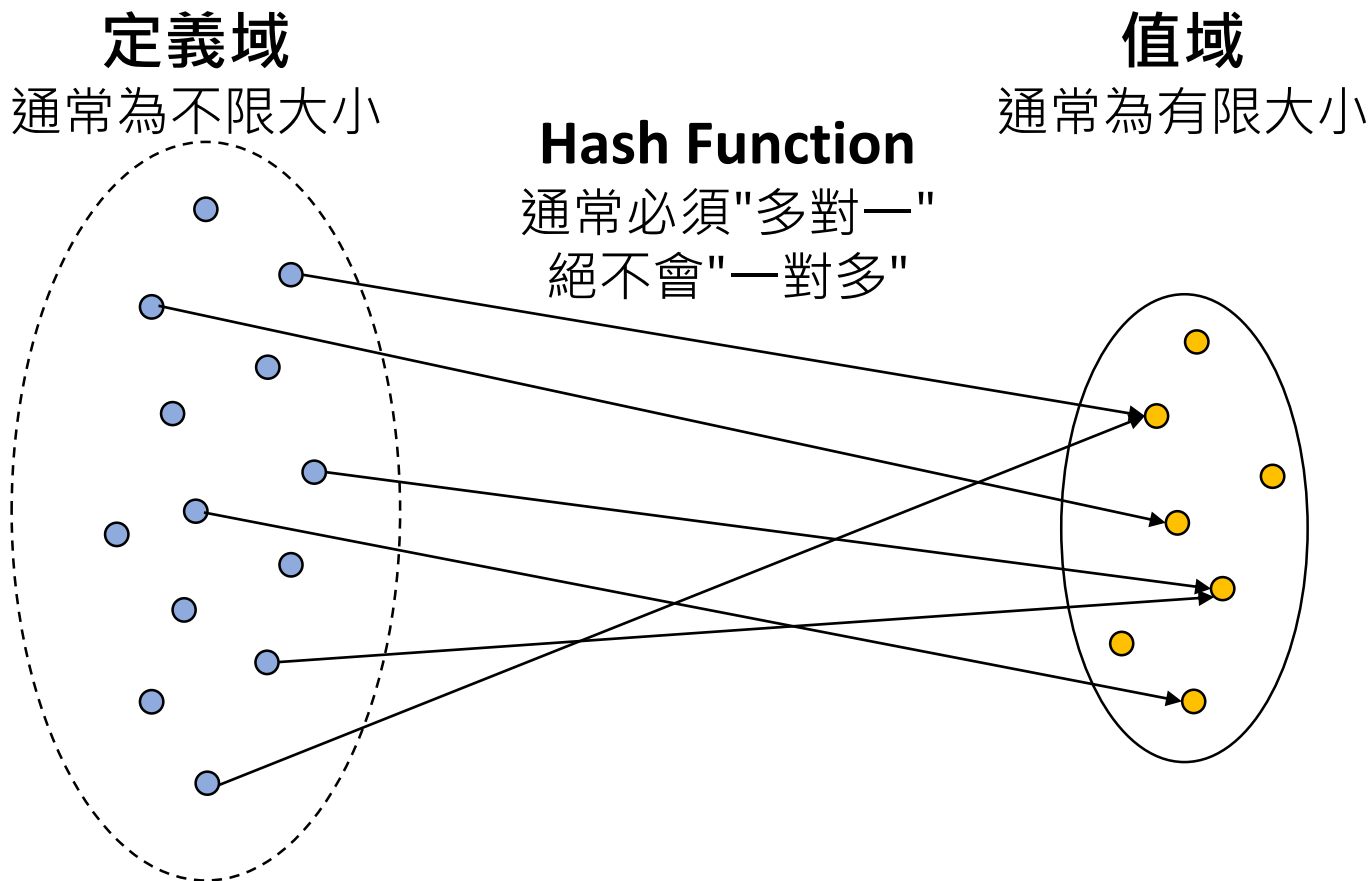
Hash Function

- Good hash functions reduce the chance of collisions

- Enlarging hash table size can also reduce collisions
  - At the cost of memory size

- Ideal hash functions
  - Uniform hash function
  - Easy to compute

Slots

A	(Alice, 100)	
B	(Bob, 80)	(Ben, 70)
⋮		
I	(Irene, 85)	
J	(John, 95)	(Jane, 100)
K	(Ken, 75)	
⋮		
Z	(Zoe, 80)	

Buckets

Hash Functions

- Classical examples
  - Modulo (division)
  - Mid-square
  - Folding
  - String-to-integer conversion

- We can design our own hash functions

Modulo (Division)

- Most widely used hash function in practice

- Procedure
  - h(k) = k % D

- Selection of D
  - D $\leq$ the number of buckets
  - D would better be an odd number
    - Even divisor D always maps even keys to even buckets and odd keys to odd buckets
    - Real-world data tend to have a bias toward either odd or even keys
  - It would be even desirable if D can be a prime number or a number having no prime factors smaller than 20

Mid-Square

- h(k) = some middle r bits of the square of k
  - The number of bucket is equal to 2^r
- Example

k	k²		h(k)
0	0	00<u>00 00</u>00	0
1	1	00<u>00 00</u>01	0
2	4	00<u>00 01</u>00	1
3	9	00<u>00 10</u>01	2
4	16	00<u>01 00</u>00	4
5	25	00<u>01 10</u>01	6
6	36	00<u>10 01</u>00	9
7	49	00<u>11 00</u>01	12

k	k²		h(k)
8	64	01<u>00 00</u>00	0
9	81	01<u>01 00</u>01	4
10	100	01<u>10 01</u>00	9
11	121	01<u>11 10</u>01	14
12	144	10<u>01 00</u>00	4
13	169	10<u>10 10</u>01	10
14	196	11<u>00 01</u>00	1
15	225	11<u>10 00</u>01	8

Folding

- Partition the key into several parts and add them together
  - Two strategies: shift folding and folding at the boundary
- Example
  - k = 12320324111220 =

123	203	241	112	20

  - Shift folding

    h(k) = ∑

123	203	241	112	20

    = 699
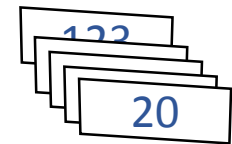
  - Folding at the boundary

    h(k) = ∑

123	302	241	211	20

    = 897
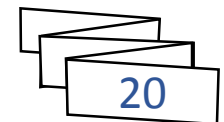
String-to-Integer Conversion

- Useful when keys are strings

- Procedure
    - Treat every n character as an 8n-bit integer
        - ASCII represents a character using 8 bits

Characters:	h	o	p	e
ASCII Values:	104	111	112	101
Binary Values:	01101000	01101111	01110000	01100101

    - Add all integers together to obtain the overall value
    - Adopt the aforementioned hash functions (modulo, folding…)
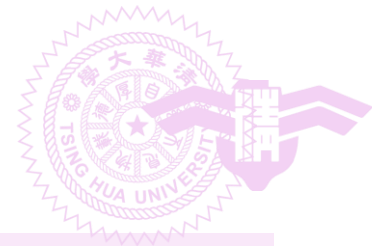
Design Our Own Hash

- Recall that
  - Hash function is **any** deterministic function that can map data of arbitrary size (original keys) to data of a desired fixed size (hashed keys)

- So of course we can design a hash like this

Original Keys → [Folding] [+ Trump's Birthday] [Mid-Square] [Modulo] → Hashed Keys
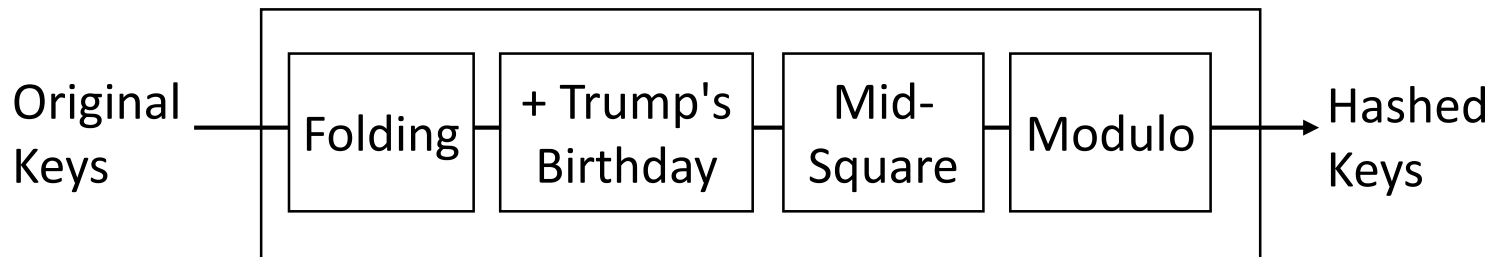
- Consideration:
  - We need to argue the advantages of our hash compared with the commonly used ones

19

Chain-Based Hash Table

- Each bucket is a chain
  - Chain nodes are typically unordered
    - We typically expect the hash function spreads records uniformly enough
    - Thus each chain does not contain too many nodes
  - Linearly traversing a chain is required for inserting, finding, and removing a key

A → (Alice, 100) | 0

B → (Bob, 80) | → (Ben, 70) | 0

⋮ | 0

I → (Irene, 85) | 0

J → (John, 95) | → (Jane, 100) | 0

K → (Ken, 75) | → (Kevin, 70) | 0

L → (Linda, 90) | 0

M → (Mary, 85) | 0

⋮ → (Zoe, 80) | 0
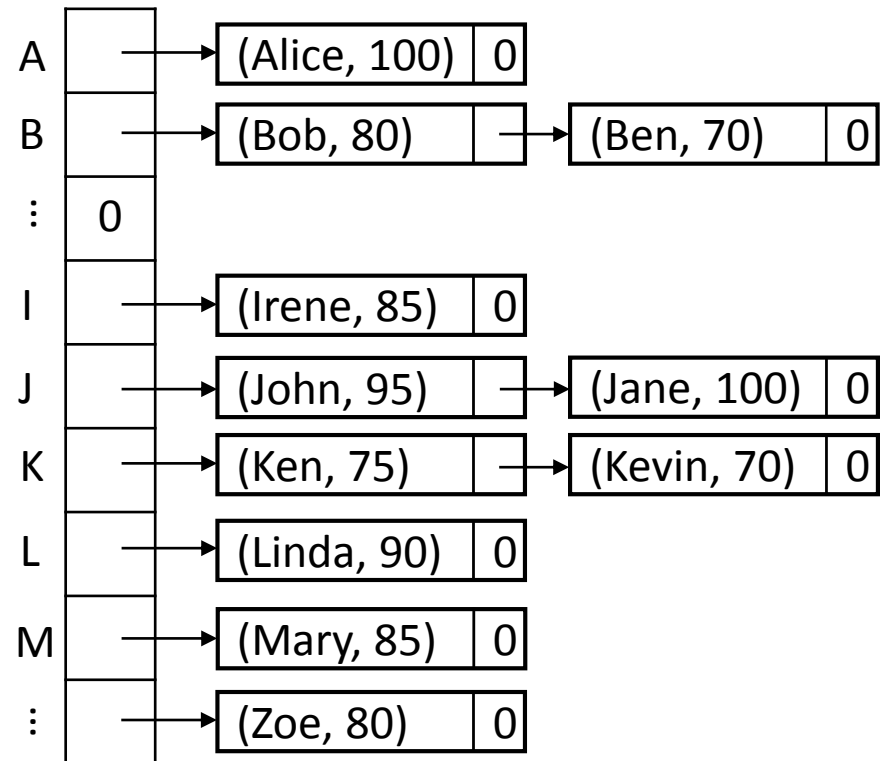
Outline

- 8.1 Introduction
- (8.2 Static hashing)
- (8.3 Dynamic hashing)
- **8.4 Bloom filters**

Bloom Filter Concepts

- Proposed by Burton Howard Bloom in 1970
- A probabilistic data structure
  - For constructing a set and then determining whether some keys is in the set

	Traditional set data structures, e.g., a BST	Bloom filters
False positive (It could be *wrong* when it says "*Yes*")	X	O (缺點)
False negative (It could be *wrong* when it says "*No*")	X	X
Easy insertion	O	O
Easy deletion	O	X (缺點)
Memory space efficiency	Low	**High** (優點)

Grocery Shop Example

- Suppose we own a grocery shop

- Customers occasionally ask for an item that we are not sure about the availability
  - We spend significant time looking for an item before realizing that the item is unavailable
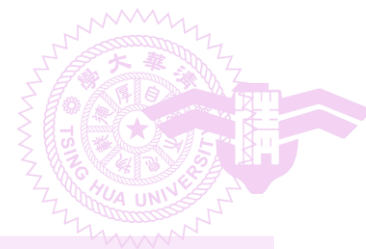
Grocery Shop Example

- Bloom filter can help
  - Determine the availability of an requested item
  - Some false positive are acceptable
    - i.e., the data structure tells that an item is available, but the fact is otherwise
  - No false negative
    - We do not want to mistakenly turn down a customer's request

Bloom Filter

- Components
  - A bit vector
  - Multiple hash functions

- Example
  - A table with 26 entries, A ~ Z
  - Three hash functions for a string
    - First character
    - Second character
    - Third character

A		N	
B		O	
C		P	
D		Q	
E		R	
F		S	
G		T	
H		U	
I		V	
J		W	
K		X	
L		Y	
M		Z	

Bloom Filter
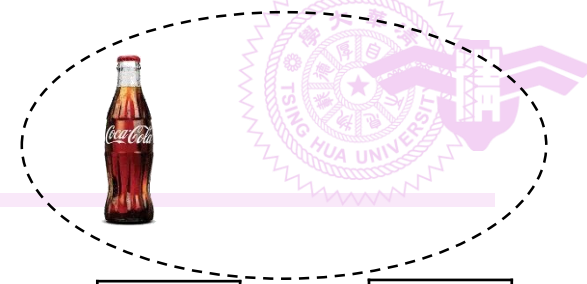
- Example
  - Register string "Coke" into the Bloom filter to indicate that our grocery sells Coke
    - Set the bit vector according to the three hash values, C, O, and K

"Coke"
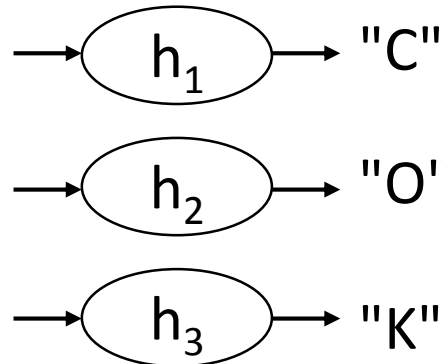
$h_1 \rightarrow$ "C"

$h_2 \rightarrow$ "O"

$h_3 \rightarrow$ "K"

A		N		
B		O	**1**	
C	**1**	P		
D		Q		
E		R		
F		S		
G		T		
H		U		
I		V		
J		W		
K	**1**	X		
L		Y		
M		Z		

Bloom Filter

- A simple test
  - If a customer request for "Coke" afterward
  - Bit vector is examined according to the three hash values
  - Bloom filter determines that coke is available because the corresponding bits have been set

"Coke"

$\rightarrow$ h_1 $\rightarrow$ "C" ✔

$\rightarrow$ h_2 $\rightarrow$ "O" ✔

$\rightarrow$ h_3 $\rightarrow$ "K" ✔

A		N	
B		O	1
C	1	P	
D		Q	
E		R	
F		S	
G		T	
H		U	
I		V	
J		W	
K	1	X	
L		Y	
M		Z	
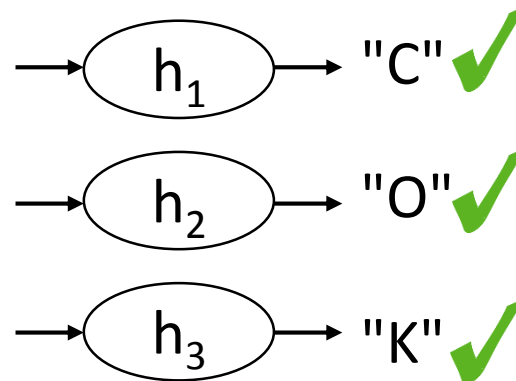
Bloom Filter

- A simple test
  - If a customer request for "orange juice" afterward
  - Bloom filter determines that orange juice is unavailable because at least one corresponding bit is not set

"Tea"

$\rightarrow$ h_1 $\rightarrow$ "T" ✗

$\rightarrow$ h_2 $\rightarrow$ "E" ✗

$\rightarrow$ h_3 $\rightarrow$ "A" ✗

A			N	
B			O	1
C	1		P	
D			Q	
E			R	
F			S	
G			T	
H			U	
I			V	
J			W	
K	1		X	
L			Y	
M			Z	
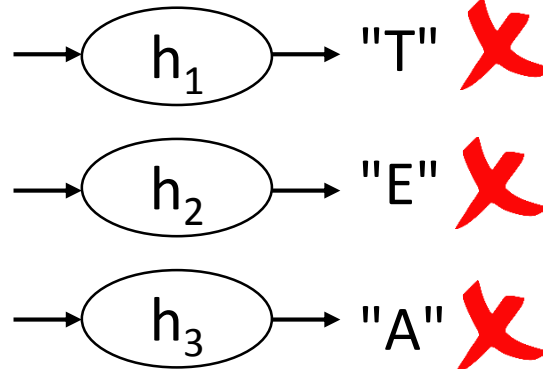
Bloom Filter

- We register more strings into the Bloom filter

"Fanta" → F A N

"Sprite" → S P R

"Vitali" → V I T

A	**1**	N	**1**
B		O	**1**
C	**1**	P	**1**
D		Q	
E		R	**1**
F	**1**	S	**1**
G		T	**1**
H		U	
I	**1**	V	**1**
J		W	
K	**1**	X	
L		Y	
M		Z	

Bloom Filter

- Test again
  - Bloom filter still works

"Coke" → C O K ✔✔✔

"Tea" → T E A ✔✘✔

"Fanta" → F A N ✔✔✔

A	**1**	N	**1**
B		O	**1**
C	**1**	P	**1**
D		Q	
E		R	**1**
F	**1**	S	**1**
G		T	**1**
H		U	
I	**1**	V	**1**
J		W	
K	**1**	X	
L		Y	
M		Z	

Advantages

- "Coca Cola
- "Fanta"
- "Sprite"
- "Vitali
  ⋮

Only 26 bits ⟶

A	**1**	N	**1**
B		O	**1**
C	**1**	P	**1**
D		Q	
E		R	**1**
F	**1**	S	**1**
G		T	**1**
H		U	
I	**1**	V	**1**
J		W	
K	**1**	X	
L		Y	
M		Z	

Disadvantages

- Bloom filter exhibits false positive
  - When Bloom filter says "yes", it is not 100% true
  - But, when Bloom filter says "no", it is always true

- "Coffee" is a false positive in our example

'Coffee" → C O F ✔✔✔

Our grocery does not sell coffee actually!

A	**1**	N	**1**
B		O	**1**
C	**1**	P	**1**
D		Q	
E		R	**1**
F	**1**	S	**1**
G		T	**1**
H		U	
I	**1**	V	**1**
J		W	
K	**1**	X	
L		Y	
M		Z	
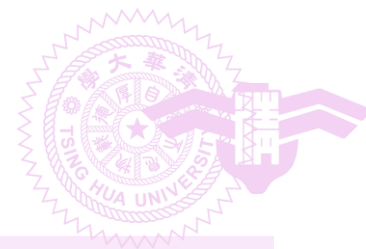
Bloom Filter Analysis

- Key factors of a bloom filter
  - Number of hash functions, k
  - Number of bits in the bit vector, m
  - Number of items expected to be stored, n
  - Uniformity of the hash functions

- False positive analysis
  - Bit vector is set nk times after n items are stored
  - Each time, the probability that a particular bit is set is ($1/m$)
    - Assume true uniformity of hash functions
  - The probability that a bit is set is ($1 - (1 - 1/m)^{nk}$) after n items are stored
  - The probability of a false positive is ($1 - (1 - 1/m)^{nk})^{k}$

- We can carefully select m, n, and k to achieve our acceptable false positive rate, e.g., 1%