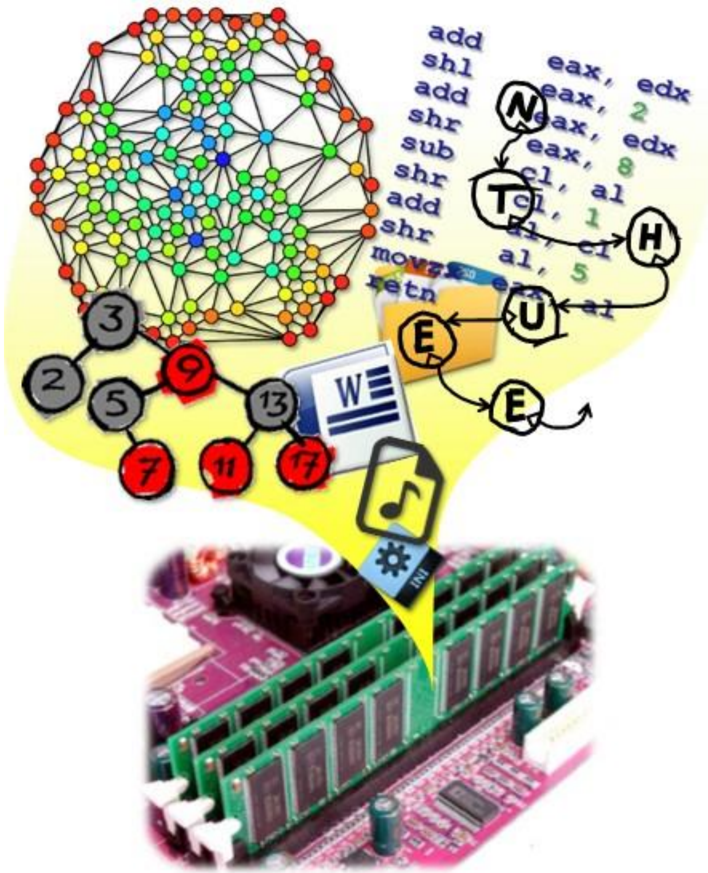
Data Structures

CH8 Hashing



Prof. Ren-Shuo Liu

NTHU EE

Spring 2017

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
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...

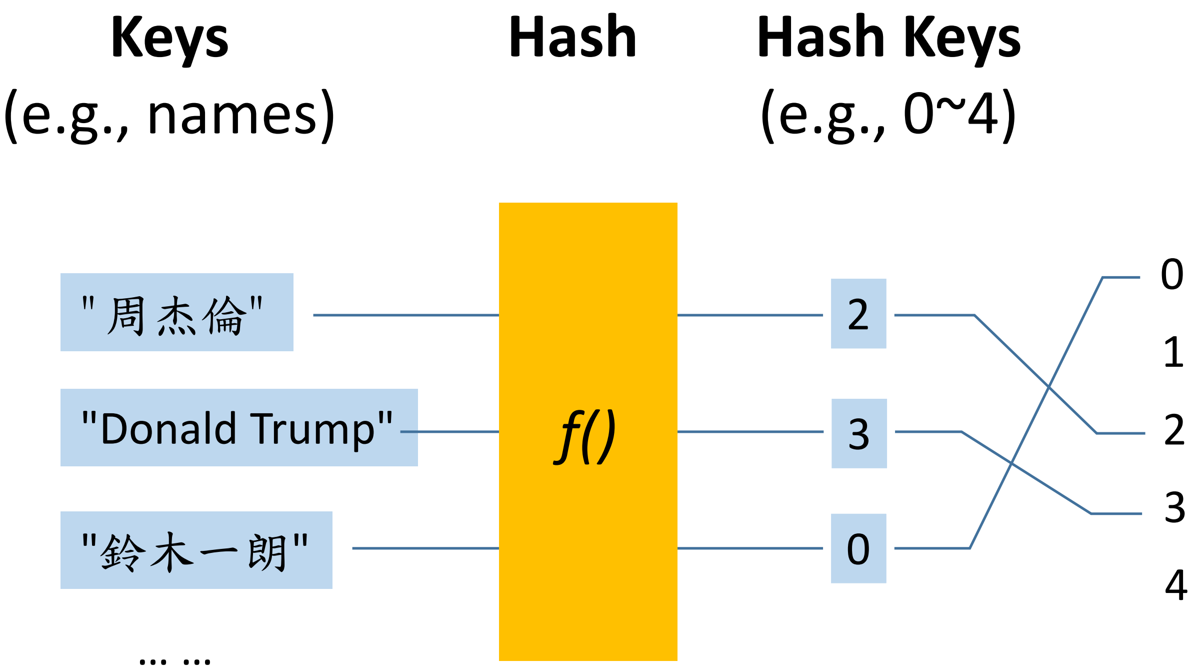
Registration Division Example

請大家向註冊組
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...

Hash Concepts

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

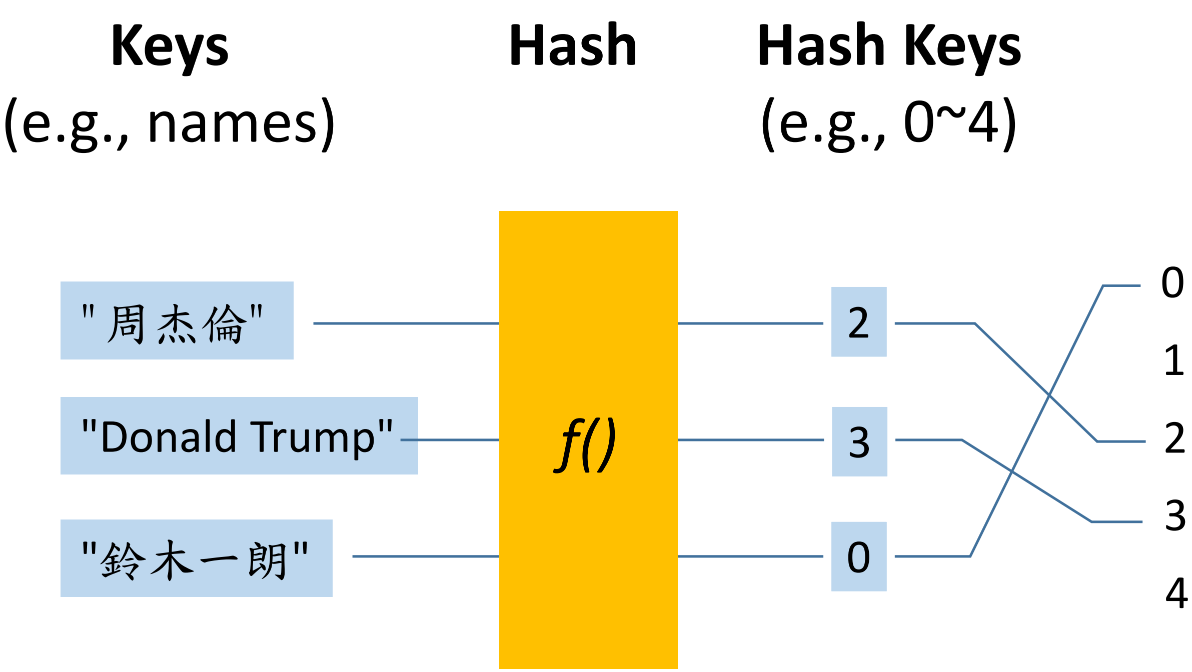


Values

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

Hash Concepts

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



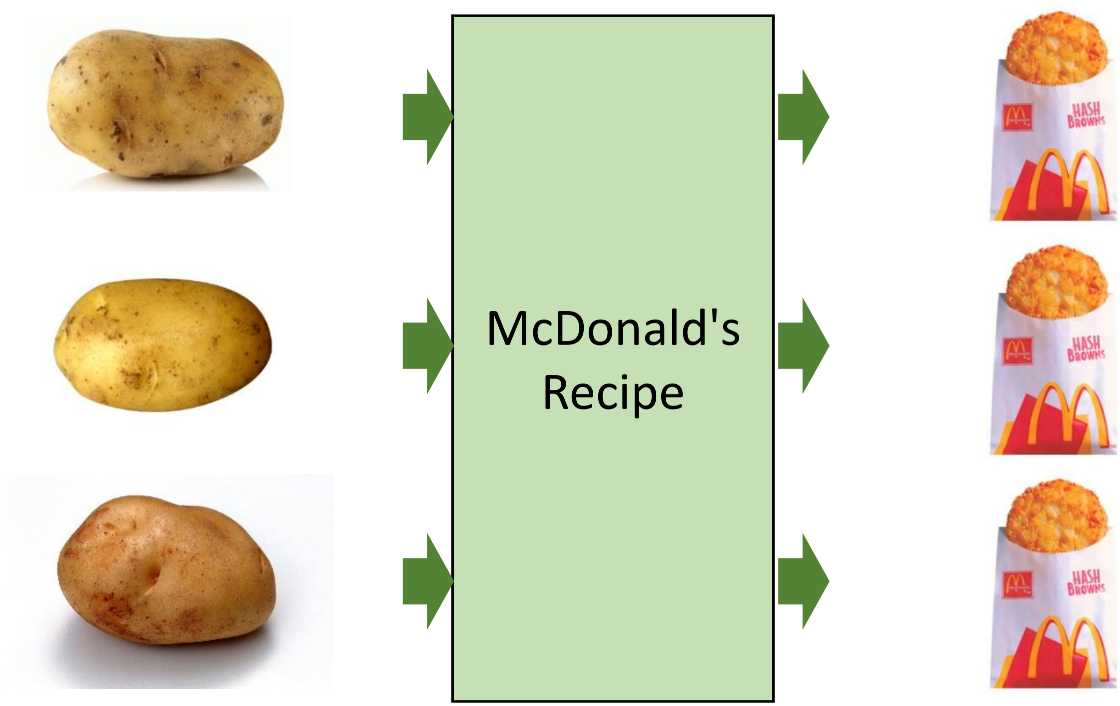
Values

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

Hash in Cooking

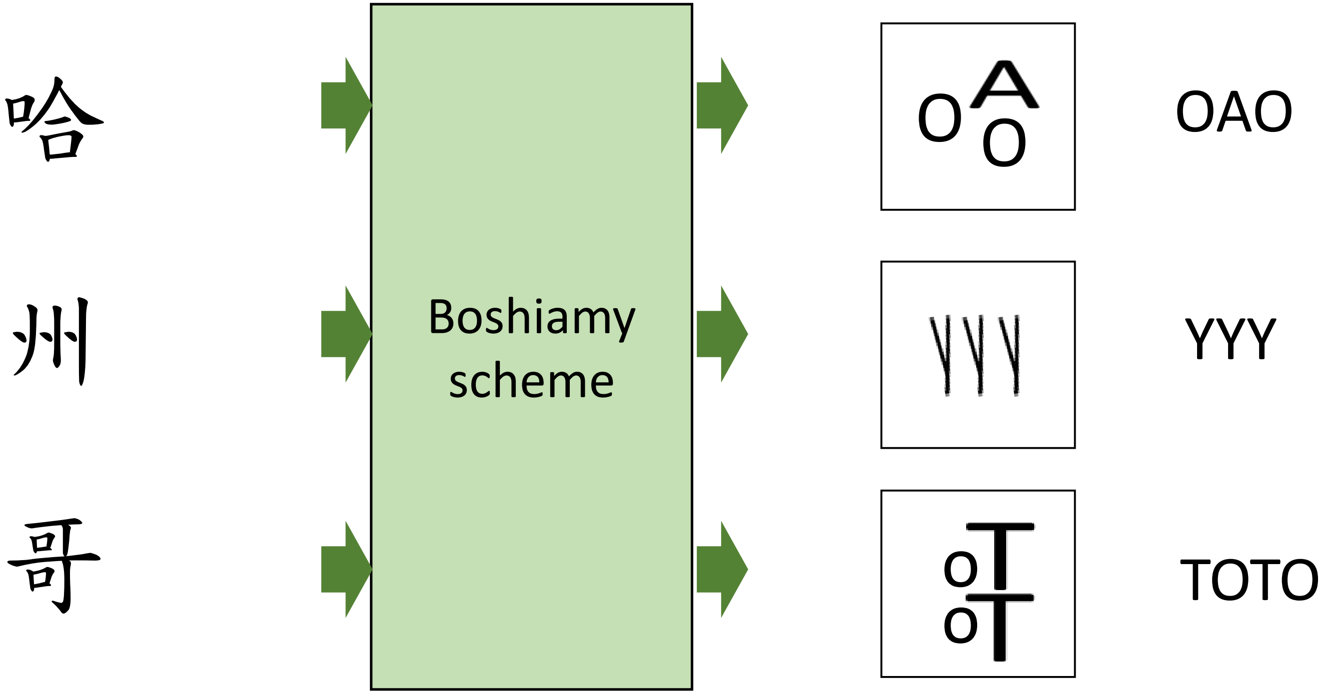
- Hash: chop and mix foods

- Example: hash browns (薯餅)



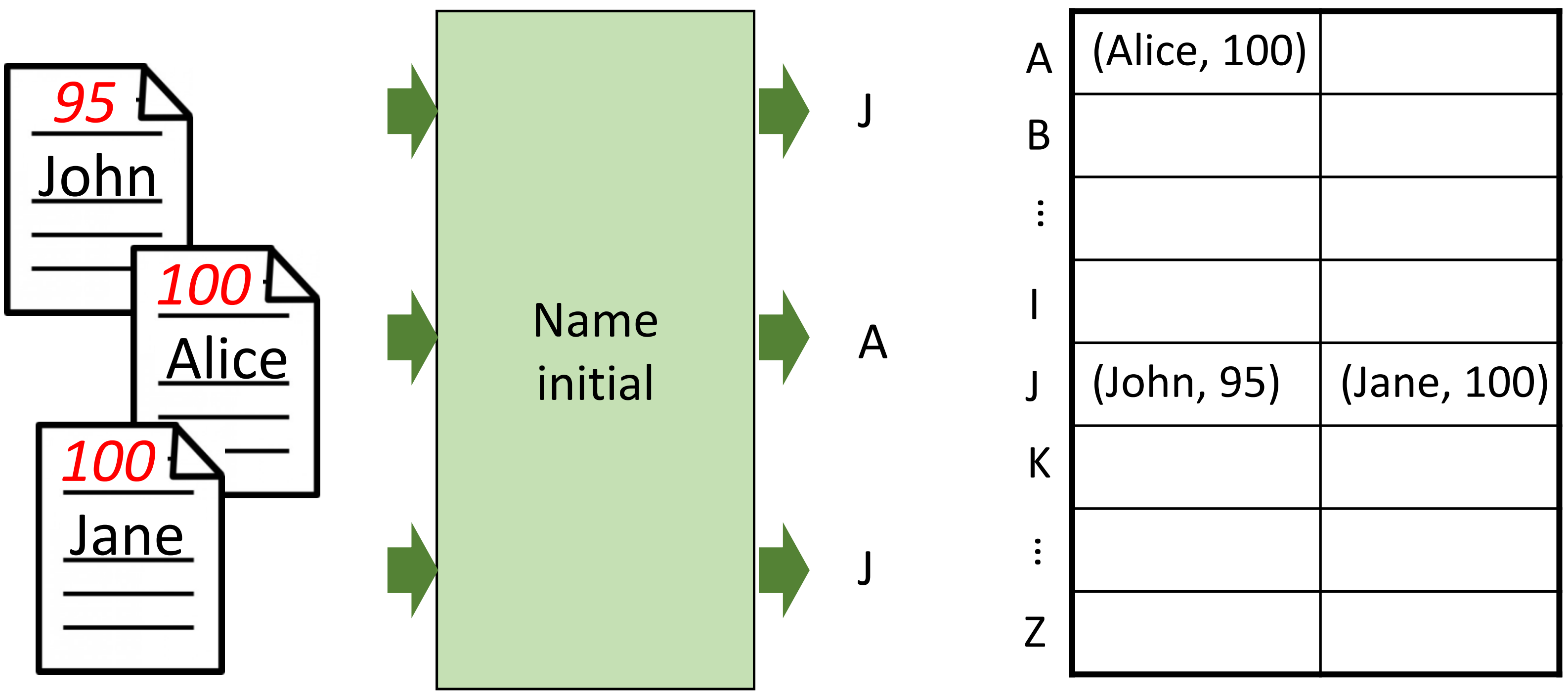
Hash in Chinese Decomposition

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



Hash in a Data Store

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



Advantages of Hashing

- Inserting, deleting, and searching can be as fast as O(1) time
  - Let hash function computation be 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

Buckets

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

Hashing

- Other terms
  - Key density ≡ n/T
  - Loading factor (or loading density) ≡ $n/(sb)$
  - k_1 and k_2 are synonyms with respect to h if $h(k_1) = h(k_2)$
  - A collision occurs when the home bucket for a newly inserted pair is non-empty
  - An overflow occurs when the home bucket for a newly inserted pair is full
- 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

Hashing

- Good hash functions reduce the chance of collisions and overflows

- Enlarging hash table size can also reduce collisions and overflows
  - To save memory, we usually do not want to do so too much

- Ideal hash functions
  - Rare collisions (i.e., a uniform hash function)
  - Easy to compute

Slots

Buckets

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

Key Techniques

- Hash functions
- Overflow handling for a hash table with a static size

Hash Functions

- Classical examples
  - Modulo (division)
  - Mid-square
  - Folding
  - Digit analysis
  - 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	k^2		h(k)
0	0	0000 0000	0
1	1	0000 0001	0
2	4	0000 0100	1
3	9	0000 1001	2
4	16	0001 0000	4
5	25	0001 1001	6
6	36	0010 0100	9
7	49	0011 0001	12

k	k^2		h(k)
8	64	0100 0000	0
9	81	0101 0001	4
10	100	0110 0100	9
11	121	0111 1001	14
12	144	1001 0000	4
13	169	1010 1001	10
14	196	1100 0100	1
15	225	1110 0001	8

Folding

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

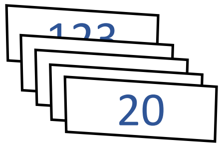
123	203	241	112	20

  - Shift folding

$h(k) = \sum$

123	203	241	112	20

= 699

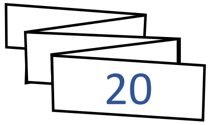


  - Folding at the boundary

$h(k) = \sum$

123	302	241	211	20

= 897



Digit Analysis

- Useful when all the keys are known in advance
- Procedure
  - Key is interpreted as a number using some radix
  - Analyze the value distributions of each digit
  - Discard digits having the most skewed distributions first
  - The remaining digits are used as the hash

k	k (radix 2)					h(k)
1	0	0	0	0	1	1
3	0	0	0	1	1	1
14	0	1	1	1	0	2
15	0	1	1	1	1	3
20	1	0	1	0	0	4
22	1	0	1	1	0	4
30	1	1	1	1	0	6
31	1	1	1	1	1	7
0:1 ratio	4:4	4:4	2:6	2:6	4:4	

String-to-Integer Conversion

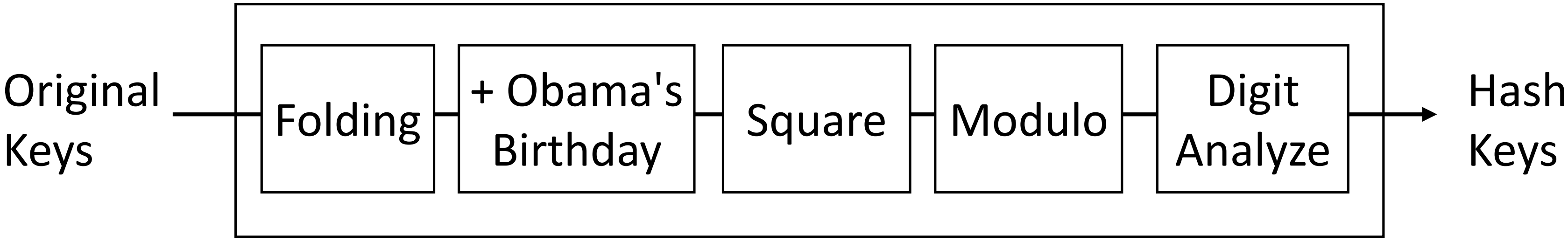
- Useful when keys are strings
- Procedure
  - Treat every n character as an 8n-bit integer
    - ASCII represents a character using 8 bits

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

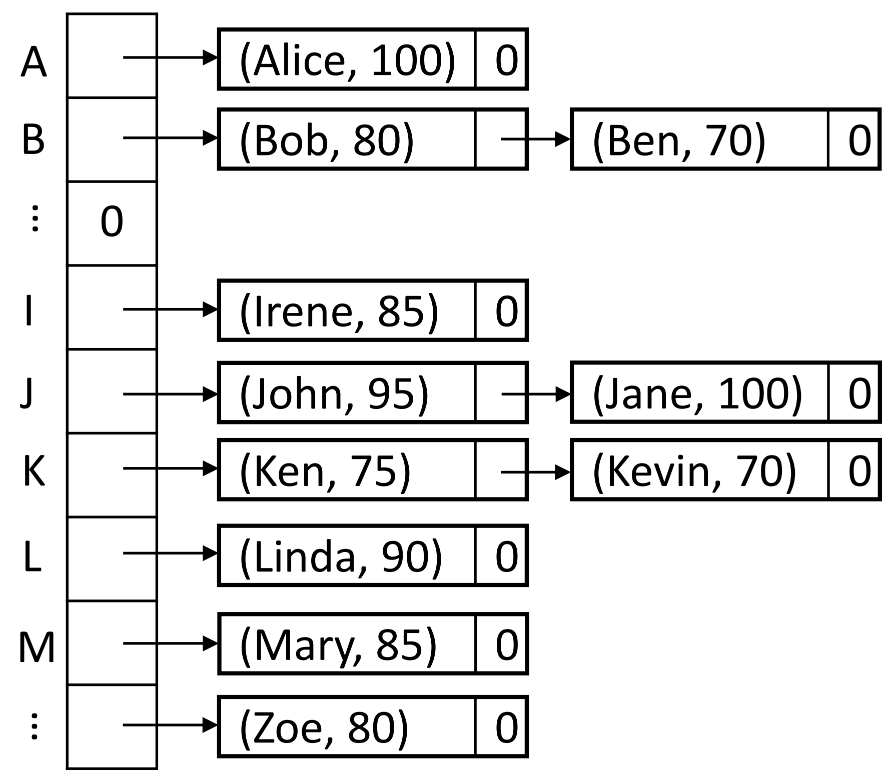
- Recall that
  - Hash function is **any** deterministic function that can map data of arbitrary size (original keys) to data of a desired fixed size (hash keys)
- So of course we can design a hash like this



- Key consideration:
  - We need to argue the advantages of our hash compared with the commonly used ones

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



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

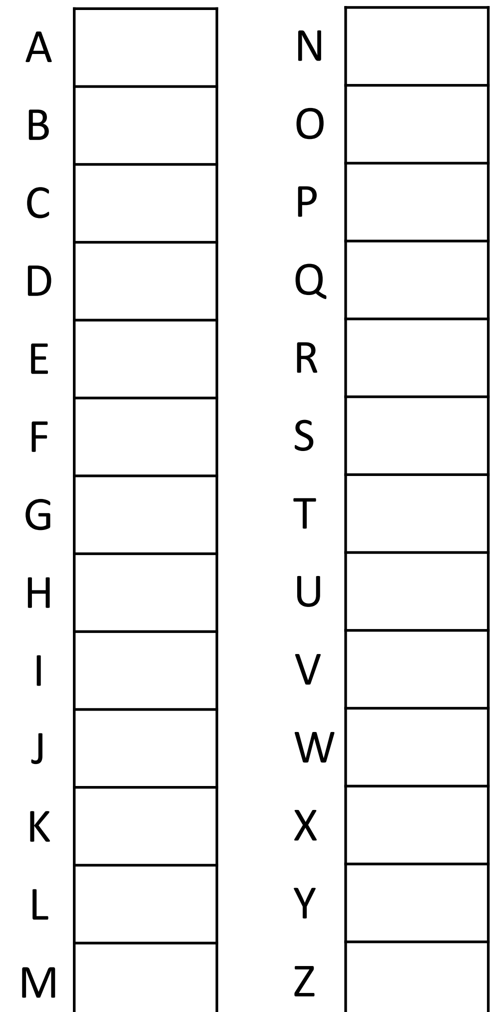
Grocery Shop Example

- Bloom filter can help
  - Determine the availability of an requested item
  - Some false positive are acceptable
    - i.e., the data structure determines 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

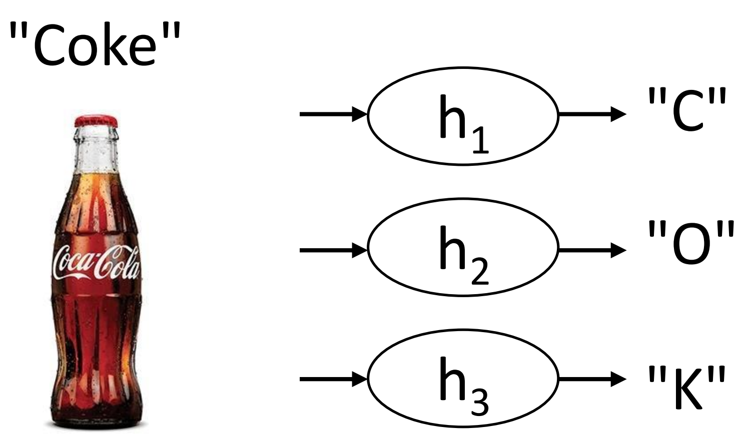
- Components
  - A bit vector
  - Multiple hash functions

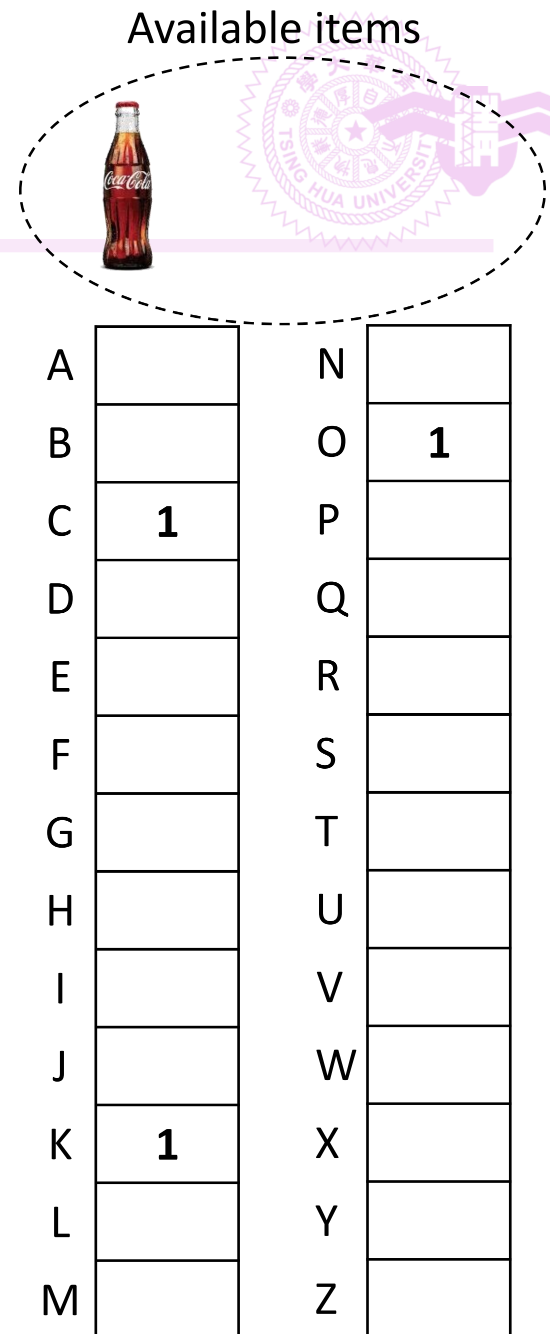
- Example
  - A table with 26 entries, A ~ Z
  - Three hash functions for a string
    - First character
    - Second character
    - Third character



Bloom Filter

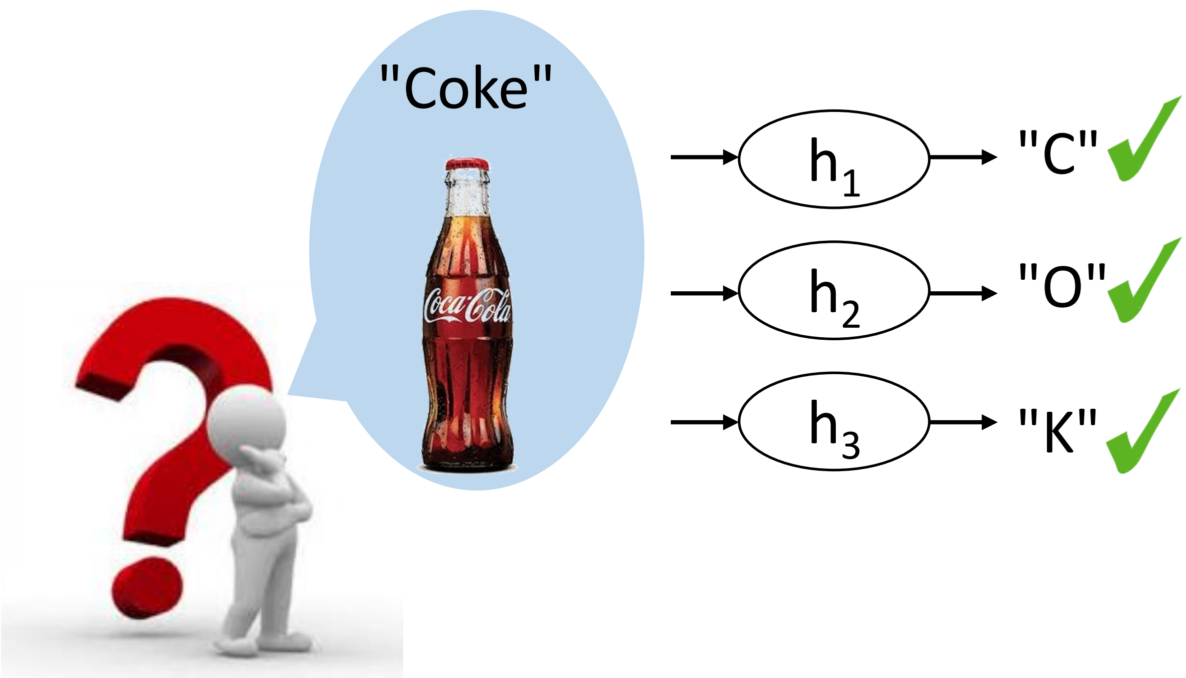
- Example
  - Register string "Coke" into the Bloom filter to indicate that our grocery sells Coke
    - Set the bit vector according to the three hash values, C, O, and K





Bloom Filter

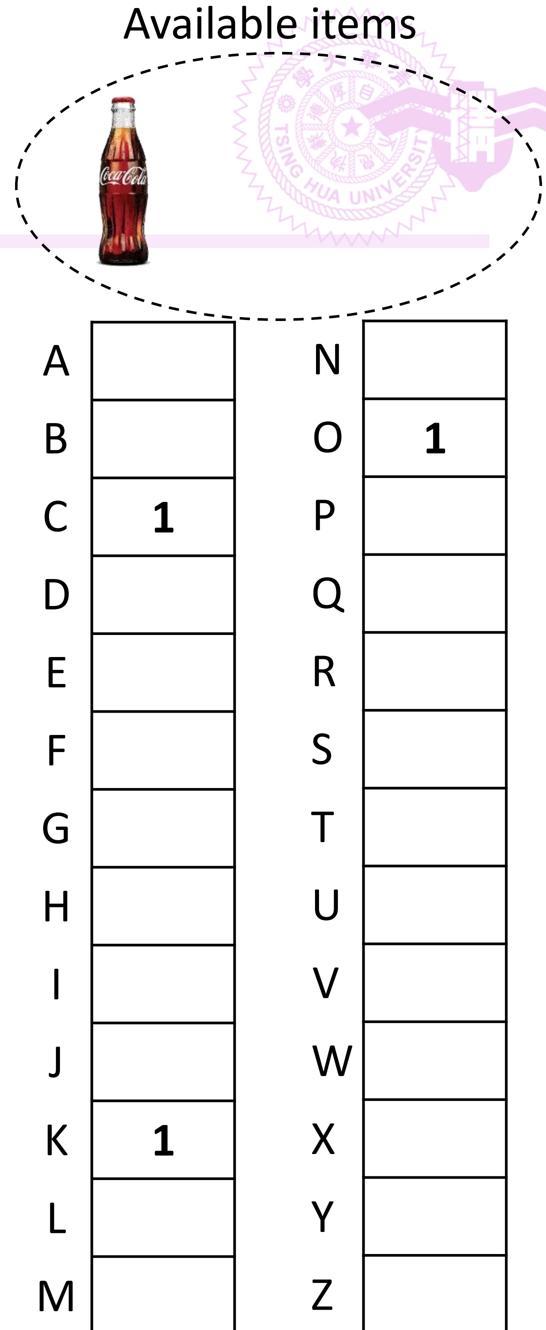
- A simple test
  - If a customer request for "Coke" afterward
  - Bit vector is examined according to the three hash values
  - Bloom filter determines that coke is available because the corresponding bits have been set

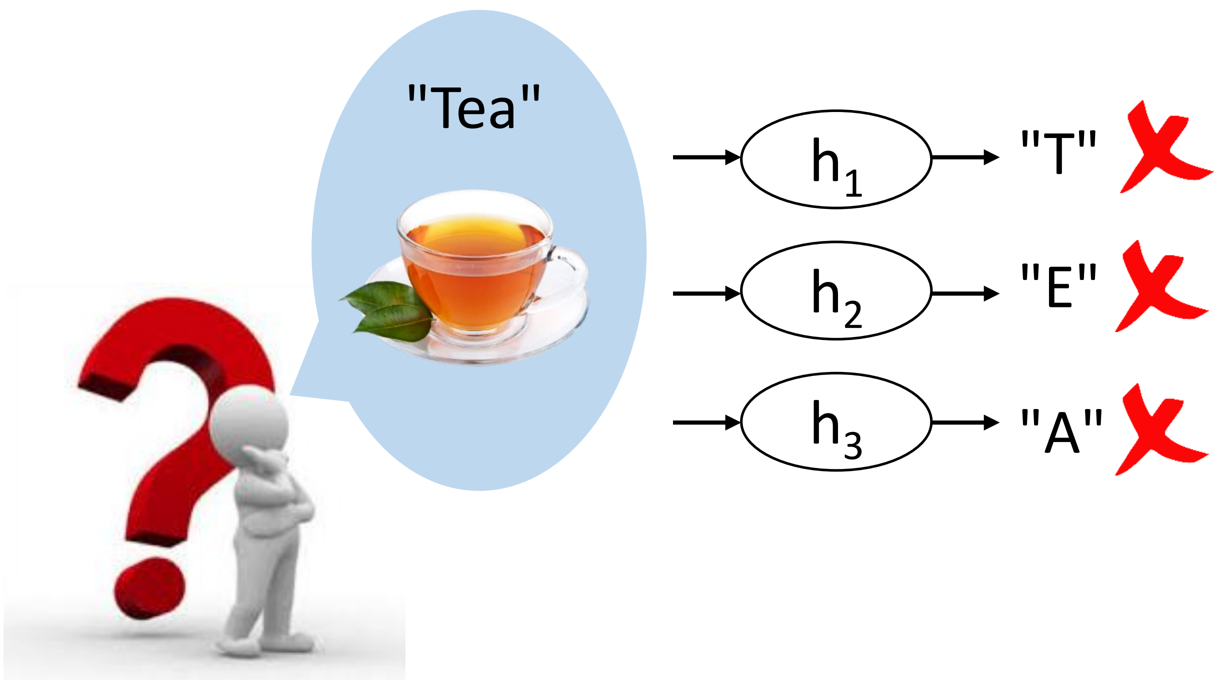


A		N	
B		O	1
C	1	P	
D		Q	
E		R	
F		S	
G		T	
H		U	
I		V	
J		W	
K	1	X	
L		Y	
M		Z	

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





Bloom Filter

Available items

- 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

Available items

- 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

Available items

- Coca Cola
- Fanta
- Sprite
- Vitali

- 26 characters (>208 bits)
- Size further grows with the number of available items

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

Available items

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	

'Coffee" → C O F

Our grocery does not sell coffee actually!

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%