HASHING

University of Waterloo

List of Slides

Other ways to Implement a Dictionary

IDEA1: Store in an *array* indexed by the *key*s

$\Rightarrow$ really fast lookups/insertions/...

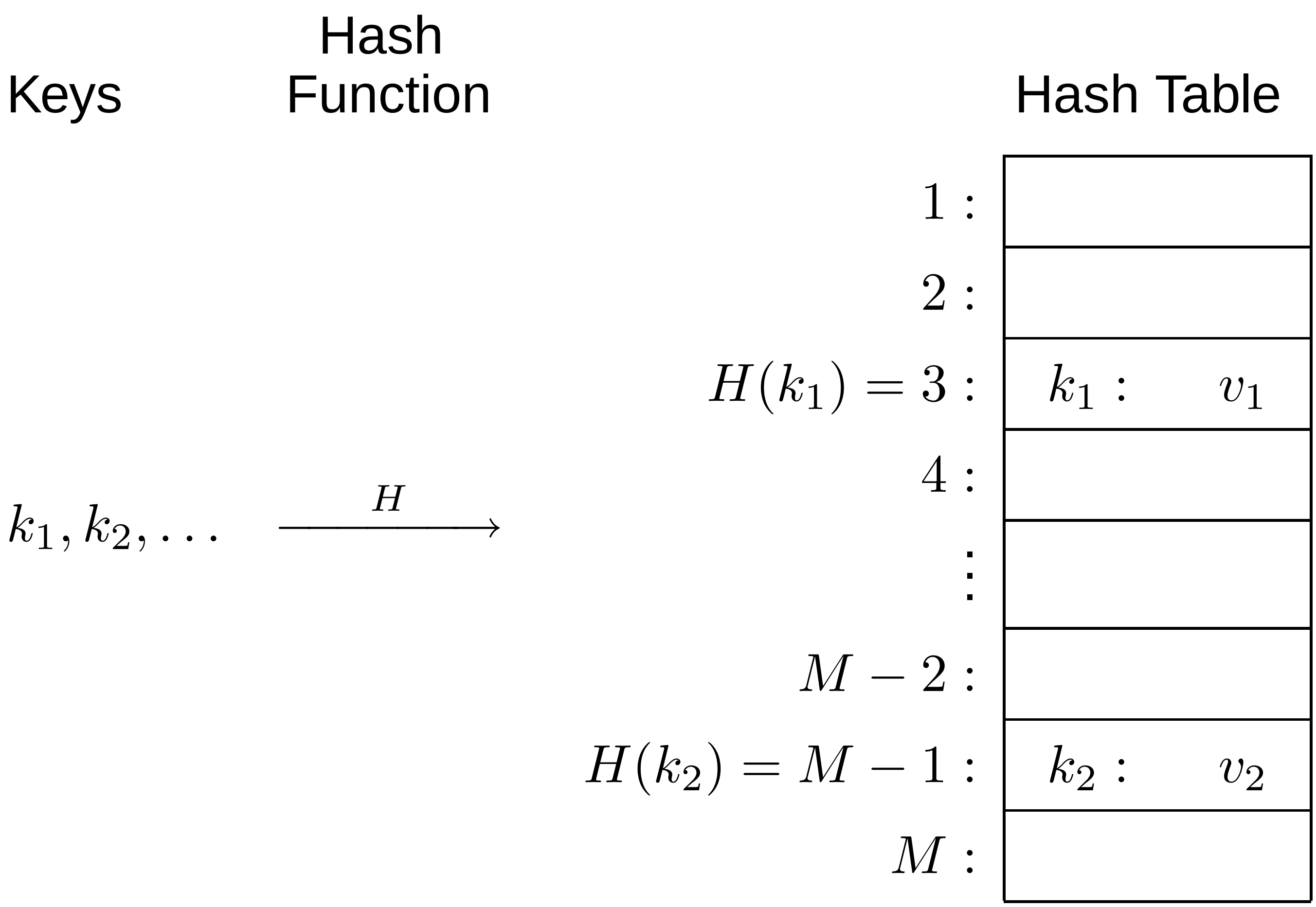
PROBLEM: VERY poor utilization of space

IDEA2: Store in an *array* indexed by results of applying a **(hash) function** H on the keys

$\Rightarrow$ requirement: $0 \leq f(k) < M$ or $0 < f(k) \leq M$ where M is the size of the **hash table**.

Example

Let H be a *hash function* with range $\{1, \ldots, M\}$:



k_i: *keys*

v_i: *values* associated with the keys

Considerations

We need to solve two problems:

1. How do we design the *hash function*?
   - static case (we know all the keys in advance):
     perfect hash functions
     $\Rightarrow$ every key is mapped to a separate value
     $\Rightarrow$ the table size is $O($number of keys$)$
   - dynamic case (we don't know the keys in advance)

     $\Rightarrow$ "mostly injective"
2. What do we do if two or more records "hash" to the same place (a **collision** occurs)?

   $\Rightarrow$ conflict (collision) resolution

Hash Functions

- Selection-based Hash Function
  Choose a part of the key to be the hash value

  $\Rightarrow$ common choices:

  - first couple of bits
  - fixed pattern of bits

  $\Rightarrow$ problems with non-uniformly distributed keys

- Division-based Hash Function
  Take the key *modulo* the table size

  $\Rightarrow$ choose table size to be *prime number*

- Folding-based Hash Function
  Take bit patterns and add them together

  $\Rightarrow$ may need an adjustment to match the table size

Collision Resolution

How good is a particular hashing scheme?

$\Rightarrow$ measured by **load factor**

$$\text{load factor} = \frac{\text{number of stored items}}{\text{size of table}}$$

The number of **probes** (accesses needed to locate a key) is compared against the load factor.

$\Rightarrow$ load factor closer to 1

$\Rightarrow$ more **collisions**

$\Rightarrow$ more **probes**

Separate Chaining

IDEA: buckets contain **list**s of elements

- use H to determine bucket
- retrieve by searching the bucket (a list!)
- insert by adding to the list

Advantages:
⇒ flexible size of buckets
⇒ deletion is easy (how?)

Disadvantages:
⇒ worst case: linear search
⇒ pointers take space

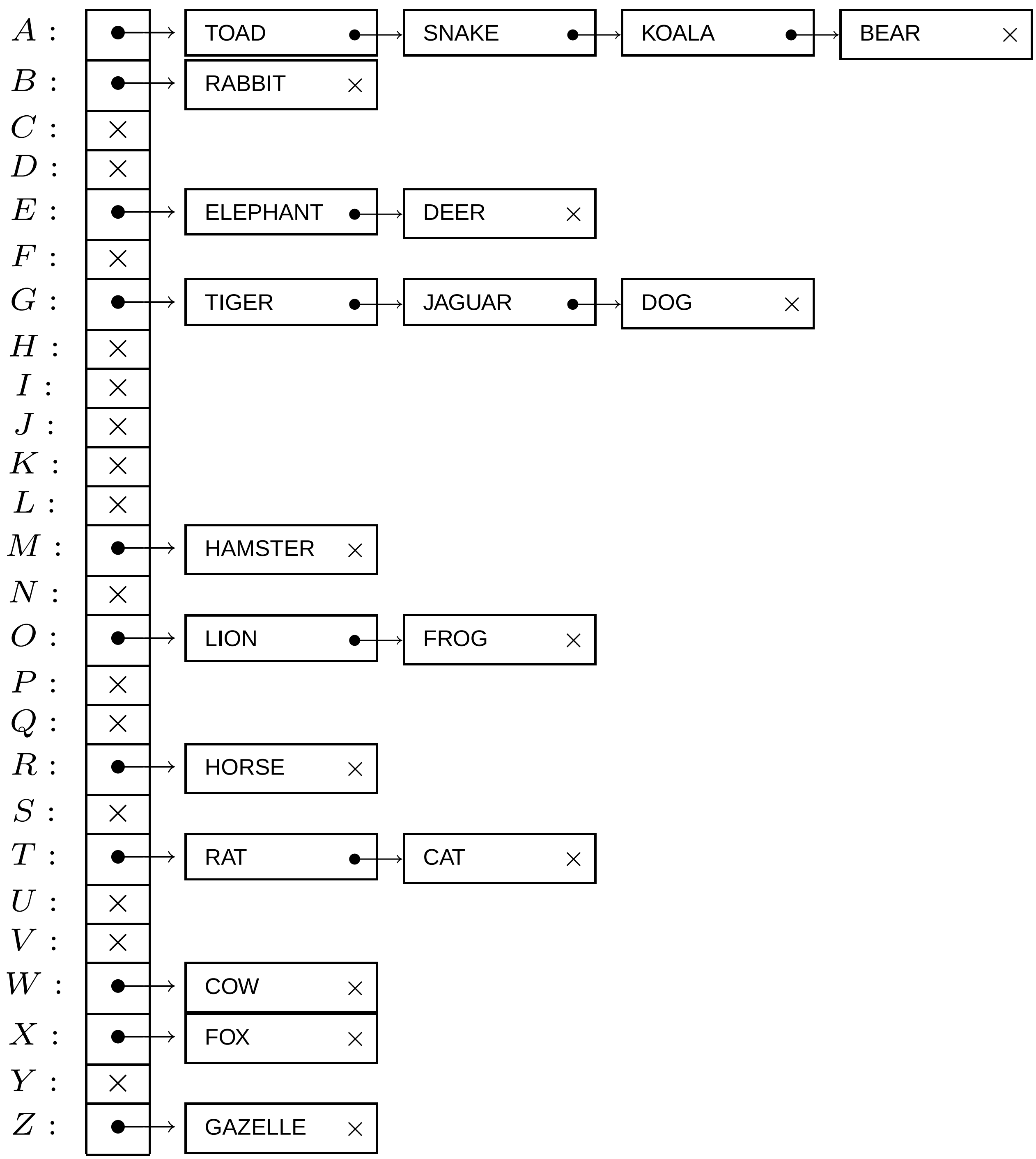
Example Data

Insert

BE**A**R	HO**R**SE
CA**T**	JA**G**UAR
CO**W**	KO**A**LA
DO**G**	LI**O**N
EL**E**PHANT	RA**B**BIT
FO**X**	RA**T**
FR**O**G	SN**A**KE
GA**Z**ELLE	TI**G**ER
HA**M**STER	TO**A**D

into a 26-bucket *Hash table* based on
a Hash function $H(k) =$ the 3rd letter of k.

Example


A :
TOAD
SNAKE
KOALA
BEAR
B :
RABBIT
C :
D :
E :
ELEPHANT
DEER
F :
G :
TIGER
JAGUAR
DOG
H :
I :
J :
K :
L :
M :
HAMSTER
N :
O :
LION
FROG
P :
Q :
R :
HORSE
S :
T :
RAT
CAT
U :
V :
W :
COW
X :
FOX
Y :
Z :
GAZELLE

Coalesced Chaining

IDEA: Store “overflow” in empty cells of the table
$\Rightarrow$ and remember where you put it
a pointer from the “original” bucket

- Lookup k:
  — look at $H(k)$,
  — if different from k follow pointers till you find it
  — or till you find a *nil* (k not found)

- Insert k:
  — similar, if not found
  — store in the next **free** cell
  — update the last pointer

Problem: deletion is quite difficult.

Example

A :	BEAR	D
B :	ELEPHANT	H
C :	JAGUAR	K
D :	KOALA	J
E :	DEER	B
F :	LION	
G :	DOG	C
H :	RABBIT	
I :	RAT	
J :	SNAKE	L
K :	TIGER	
L :	TOAD	
M :	HAMSTER	
N :		
O :	FROG	F
P :		
Q :		
R :	HORSE	
S :		
T :	CAT	I
U :		
V :		
W :	COW	
X :	FOX	
Y :		
Z :	GAZELLE	

Open Addressing

IDEA don't use pointers

⇒ use *fixed* **probe sequence**
⇒ $H(k,i)$ is a hash function used for the i-th probe
⇒ $H(k,i)$ should *cover* the whole table for varying i

Most common version: **Sequential (linear) probing**:

$$\begin{aligned} H(k,1) &= H(k) \\ H(k,i+1) &= (H(k,i)+1)) \bmod M \end{aligned}$$

Problem: *primary clustering*

⇒ "clogging" parts of the table
⇒ long chains of *probes*
to insert "TOAD" we need 10 probes

⇒ using different "offset" doesn't help
and may miss parts of the table
one reason to pick M prime

Another problem: deletion

⇒ only possibility: mark as deleted.

Example

A :	BEAR
B :	KOALA
C :	RABBIT
D :	SNAKE
E :	DEER
F :	ELEPHANT
G :	DOG
H :	JAGUAR
I :	TIGER
J :	TOAD
K :	
L :	
M :	HAMSTER
N :	
O :	FROG
P :	LION
Q :	
R :	HORSE
S :	
T :	CAT
U :	RAT
V :	
W :	COW
X :	FOX
Y :	
Z :	GAZELLE

Double Hashing

IDEA: use 2nd hash function to determine the probe sequence for k

$$
\begin{aligned}
H(k,1) &= H_1(k) \\
H(k,i+1) &= (H(k,i) + H_2(k)) \bmod M
\end{aligned}
$$

$\Rightarrow H_2(k)$ must be relatively prime to M!

In our Example:

$H_2(k) = i$ if the first letter of k is i^{th} in alphabet.

Example

A :	BEAR
B :	RABBIT
C :	
D :	
E :	DEER
F :	
G :	DOG
H :	
I :	RAT
J :	ELEPHANT
K :	TIGER
L :	KOALA
M :	HAMSTER
N :	
O :	FROG
P :	
Q :	JAGUAR
R :	HORSE
S :	
T :	CAT
U :	TOAD
V :	
W :	COW
X :	FOX
Y :	
Z :	GAZELLE
0 :	LION
1 :	
2 :	SNAKE

Probe Sequences

Key	**Probes**
BEAR	A,C,E,G,. . .
CAT	T,W,Z,2,. . .
COW	W,Z,2,C,. . .
DOG	G,K,O,S,. . .
ELEPHANT	E,J,O T,. . .
FOX	X,A,G,M. . .
FROG	O,U,0,D,. . .
GAZELLE	Z,D,J,P,. . .
HAMSTER	M,U,2,H,. . .
. . .	

Summary

- Fast array-like search

  $\Rightarrow$ degenerates for high load factors ($> .7$)

- Easy implementation

  $\Rightarrow$ no pointer overhead

- Many versions

  $\Rightarrow$ extensible/linear hashing

  the table grows/shrinks with inserts/deletes

  disk (block-based) versions