Hash Tables

Implement the MAP ADT

Hash functions and hash tables

- Idea and Examples

Hash function details

- Address Generation (Hash code + Compression code)
- Collision Resolution
  - Linear probing
  - Quadratic probing
  - Double hashing

Idea

Hash tables are data structures that implement a MAP ADT

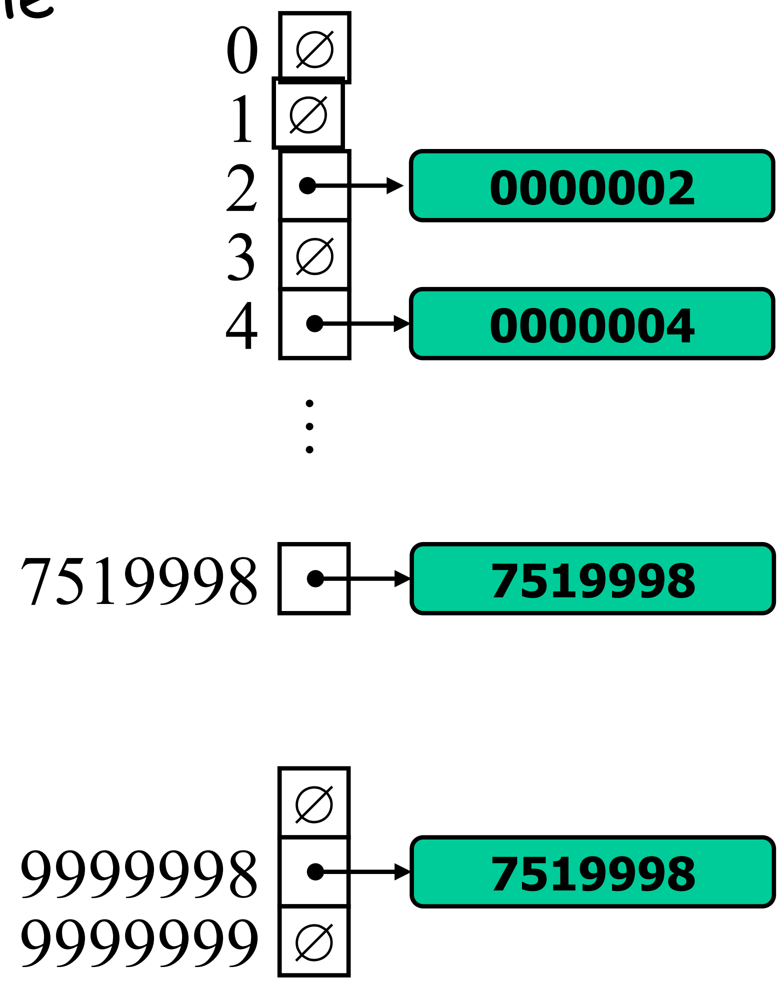
Data is stored and retrieved by use of a function of the key.

It is stored, but not sorted!

Example

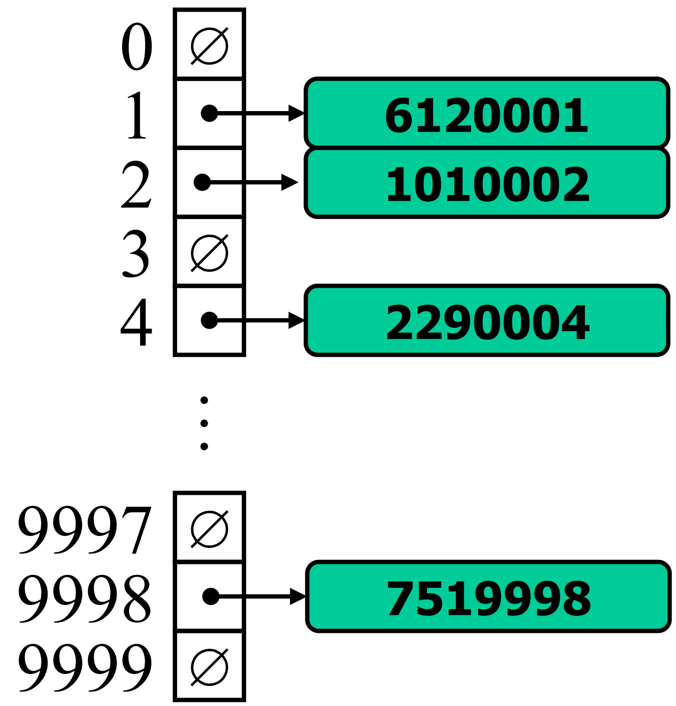
Student records are stored in an array using a 7 digit student i.d. the index.

If the i.d. were used unmodified, the array would have to have enough room for 10,000,000 student records.



Example

Instead, student i.d.'s are "hashed" to produce an integer between, say 1 and 10,000 which indexes into an array.



Problem

Since a possible 10,000,000 numbers are being compressed into just 10,000 how can we guarantee that no 2 i.d.'s end up stored in the same place?

Problem A

Address Generation

Construction of the function $h(K_i)$

- **Simple to calculate**
- **Uniformly distribute the elements in the table**

Problem B

Collision Resolution

What strategy to use if two keys map to same location h(Ki)

The general Idea:

$\forall$ key K_i

$h(K_i)$ = position of K_i in the table

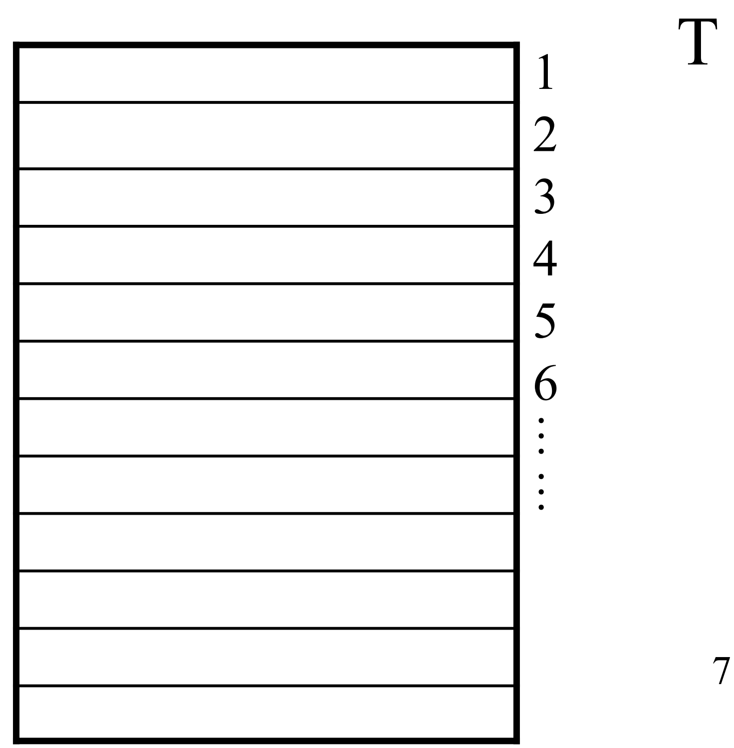
$h(K_i) = pos$ pos: integer

$h(K_i) \neq h(K_g)$ $i \neq g$

Search for key K_i

O(1) !

Insertion



—Example —

The keys all have different first letters.

CAT, ELEPHANT, FOX,

SKUNK, ZEBRA

h (CAT) = 2
h (ELEPHANT) = 4
h (FOX) = 5
⋮

	0
	1
CAT	2
	3
ELEPHANT	4
FOX	5
⋮	
SKUNK	
⋮	
ZEBRA	

Problem

If we want to insert a key that doesn't have a different first letter

COLLISIONS

	0
	1
CAT	2
	3
ELEPHANT	4
FOX	5
	6
	7
	8
⋮	⋮
SKUNK	
⋮	
ZEBRA	

Problem

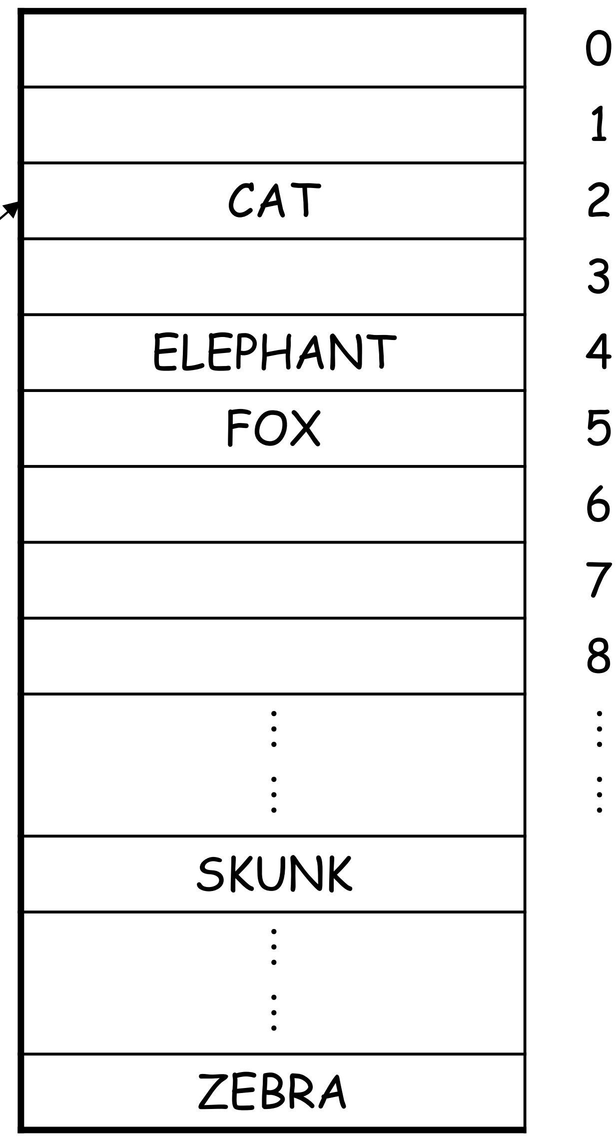
If we want to insert a key that doesn't have a different first letter

COLLISIONS

We want to insert:
CRICKET

h (CRICKET) = 2

index 2 is occupied



Definition:

load factor of an Hash Table

$$\boxed{\alpha = \frac{n}{N}}$$

\# of elements
\# of cells

Address Generation

Split problem into 2 sub-problems:

Hash code map:

h_1: keys $\rightarrow$ integers

$$\mathbf{h(x) = h_2(h_1(x))}$$

Compression map:

h_2: integers $\rightarrow$ [0, **TableSize** - 1]

Hash Code Maps

Hash codes reinterpret the key as an integer. They
need to: 1. Give the same result for the same key
and should: 2. Provide good "spread"

Examples:

- Memory address:
  - We reinterpret the memory address of the key object as an integer (default hash code of all Java objects)
- Integer cast:
  - We reinterpret the bits of the key as an integer
- Component sum:
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)

Hash Code Maps (cont.)

- Polynomial accumulation:
  - We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)

    $a_0\ a_1\ \ldots\ a_{n-1}$
  - We evaluate the polynomial

    $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$

    at a fixed value z, ignoring overflows
  - Especially suitable for strings (e.g., the choice $z = 33$ gives at most 6 collisions on a set of 50,000 English words)

Compression Maps

Compression Maps:

• Take the output of the hash code and compress it into the desired range.

• If the result of the hash code was the same, the result of the compression map should be the same.

•Compression maps should maximize "spread" so as to minimize collisions.

Compression Maps Examples

- Division:
  - $h_2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime (number theory).

- Multiply, Add and Divide (MAD):
  - $h_2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Address Generation
Some examples …

Address Generation (a)

000000000
000000001
000000011

....

N = size of the table

$r = \lceil \log N \rceil$

Example: $N = 2^9$

r=9 : number of bits to represent a cell

For a given key, the Hash code must return a sequence of bit

The Compression Map must return 9 bits representing a cell

Address Generation (a)

N = size of the table

r = $\lceil \log N \rceil$

Hash code

$h_1(x)$: gives a binary string

Compression Map

a) $h_2(h_1(x))$: = subset (of r bits) of $h_1(x)$

a.1) the r least significant bits

a.2) the r most significant bits

a.3) the central r bits

→ Simple to calculate

→ Doesn't guarantee a random distribution

—Example —

Keys are words (animals)
Each word: 6 characters (just for this example)

h_1 transforms the key into a string of bits

Coding of letters

A	000001
B	000010
C	000011
⋮	
H	001000
⋮	
T	010010
⋮	
-	100000

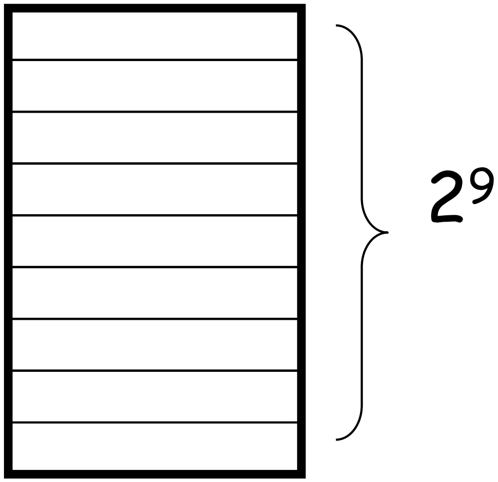
<u>**CHAT--**</u>

$$\underbrace{000011}_{C}\ \underbrace{001000}_{H}\underbrace{000001}_{A}\underbrace{010010}_{T}\underbrace{100000}_{-}\underbrace{100000}_{-}$$

<u>**BAT---**</u>

$$000010000001010010100000100000100000$$

Size of the table: $N = 2^9$



$N = 2^9$
$r = 9$

<u>CHAT--</u>

$h_2(0000011001000000000101001010100000100000) =$

C H A T - -

a1) Example of address Generation: the r least significant bits

(r = 9)

$h_2(000011001000000001010010100000100000) =$
000100000

–

All the animals of 4 (or less) characters hash to the same location.

a2) Example of address Generation: the r most significant bits

(r = 9)

$h_2(000011001000000000101001010000000100000) =$
000011001

All the animals that begin with the same first two letters hash to the same location.

Address Generation (b)

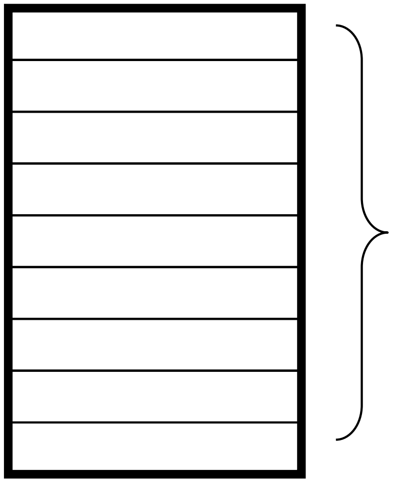
$h_1(x)$: gives a binary string

b) $h_2(h_1(x))$: sum of subset of bits of $h_1(x)$

→ Simple to calculate

→ More random than **a)**

—Example —



2^9

$N = 2^9$
$r = 9$

<u>CHAT--</u>

Coding of letters

A	000001
B	000010
C	000011
⋮	
H	001000
⋮	
T	010010
⋮	
␣	100000

$h_2(\underline{000011001}00000\underline{000101001}0100\underline{000100000}) =$

000011001 most significant
000101001 central
<u>000100000</u> least significant

XOR 000010000

Address Generation (c)

$h_1(x)$: gives a binary string

c) $h_2(h_1(x))$: subset (of r bits) of $h_1(x)^2$

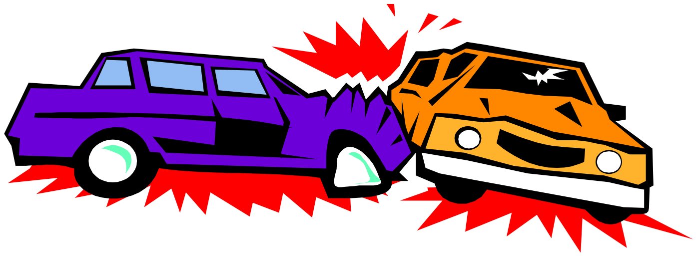
→ Multiplication is involved

→ More random than **a)** and **b)**

Address Generation (d)

d) $h_2(h_1(x)) := h_1(x)$ MOD N

→ Division is involved!
→ Very random (if N is odd)

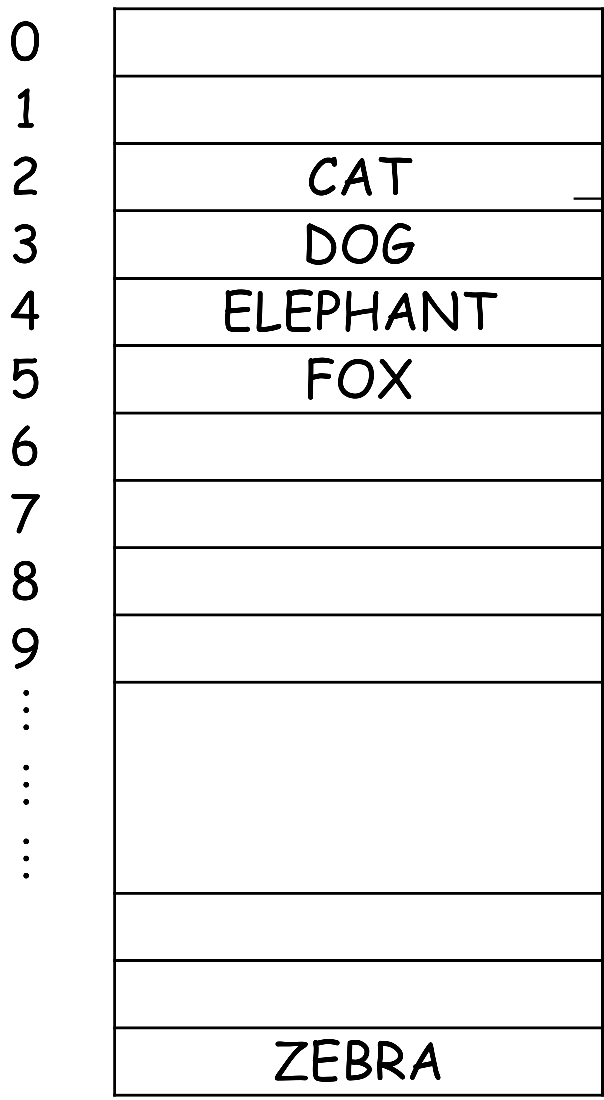


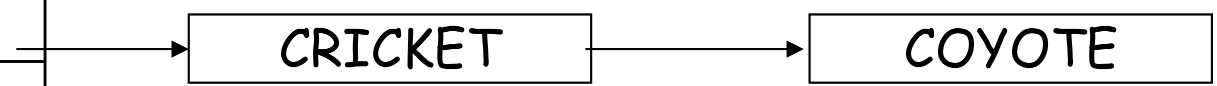
Collision Resolution

Collision Resolution

Separate Chaining

Ex: cricket, cccc





- findElement(***k***)
- removeElement(***k***)
- insertItem(k,o)

Collision Resolution (examples)

1. Open Addressing

0	
1	
2	CAT
3	CRICKET
4	ELEPHANT
5	FOX
6	
7	
8	
9	
⋮	
	ZEBRA

→COYOTE

- h (COYOTE) = 2 OCCUPIED
- We consider 3 OCCUPIED
- We consider 4 OCCUPIED
- " 5 OCCUPIED
- " 6 FREE!

Linear Probing

Collision Resolution (1)
Linear Probing

$$\underbrace{h(K_i)}_{h_0(K_i)},\ \underbrace{h(K_i)+1}_{h_1(K_i)},\ \underbrace{h(K_i)+2}_{h_2(K_i)},\ \underbrace{h(K_i)+3}_{h_3(K_i)}\ \ldots.$$

Let $h_0(K_i) = h(K_i)$

$$\boxed{h_j(K_i) = [h(K_i) + j] \bmod N}$$

Search with Linear Probing

- Consider a hash table A that uses linear probing
- findElement(k)
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

Algorithm *findElement*(k)
    $i \leftarrow h(k)$
    $p \leftarrow 0$
    repeat
        $c \leftarrow A[i]$
        if $c = \varnothing$
            return *NO_SUCH_KEY*
        else if *c.key* () $= k$
            return *c.element*()
        else
            $i \leftarrow (i + 1) \bmod N$
            $p \leftarrow p + 1$
    until $p = N$
    return *NO_SUCH_KEY*

Updates with Linear Probing

- To handle insertions and deletions, we introduce a special object, called **AVAILABLE**, which replaces deleted elements
- removeElement(***k***)
  - We search for an item with key ***k***
  - If such an item (***k***, ***o***) is found, we replace it with the special item ***AVAILABLE*** and we return element ***o***
  - Else, we return ***NO_SUCH_KEY***
- insert Item(***k***, ***o***)
  - We throw an exception if the table is full
  - We start at cell ***h***(***k***)
  - We probe consecutive cells until one of the following occurs
    - **A cell *i* is found that is either empty or stores AVAILABLE, or**
    - ***N* cells have been unsuccessfully probed**
  - We store item (***k***, ***o***) in cell ***i***

Performances of Linear Probing

Search: Average number of probes ....

$C(\alpha)$

Experimental results for a hash table with load factor α

$\alpha = n/N$	$C(\alpha)$
0.1 (10%)	1.06
0.5 (50%)	1.50
0.75 (75%)	2.50
0.9 (90%)	5.50

Example of Linear probing

N = 13

$$h_j(k_i) = [h(k_i) + j] \; mod \; N$$

Insert keys 13,15,5,28,1,14,2 in this order

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

⇩

13	1	15	28	14	5	2						
0	1	2	3	4	5	6	7	8	9	10	11	12

Problem
with Linear Probing: PRIMARY CLUSTERING

2	CAT
3	CRICKET
4	ELEPHANT
5	FOX
6	CCC
7	

h (COYOTE) $= 2$
h_1 (COYOTE) $= 3$
$h_2 = 4$
$h_3 = 5$
$h_4 = 6$
$h_5 = 7$!

Here we are using as address generation the integer corresponding to the first letter

Idea:

Use a non-linear probe

Collision Resolution (2)

Quadrating Probing

$$\underbrace{h(k_i)}_{h_0(k_i)}, \underbrace{h(k_i)+1}_{h_1(k_i)}, h(k_i)+4, h(k_i)+9, \ldots$$

$$\boxed{h_j(k_i) = [h(k_i) + j^2] \ mod \ N}$$

N: prime

→ *mod* is hard to calculate

→Visits only half of the table

but…

Example of Quadratic probing

$N = 13$

$$h_j(k_i) = [h(k_i) + j^2] \; mod \; N$$

Insert keys 13,15,5,28,1,14,2 in this order

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

⇩

13	1	15	28		5	2				14		
0	1	2	3	4	5	6	7	8	9	10	11	12

13	1	15	28	-	5	2	-	-	-	14	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12

Find 14:

$h_0(14) = 14 \bmod 13 = 1$

Probe 1

Probe 2

Probe 5

Probe 10 --- FOUND

13	1	15	28	-	5	2	-	-	-	14	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12

Delete 5:

Probe 5

13	1	15	28	-	avail	2	-	-	-	14	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12

13	1	15	28	-	avail	2	-	-	-	14	-	-
0	1	2	3	4	5	6	7	8	9	10	11	12

Find 14:

Probe 1

Probe 2

Probe 5

???

Probe 10

Performances of Quadratic Probing

Experimental results for a hash table with load factor α

Search

$\alpha = n/N$	$C(\alpha)$
0.1 (10%)	1.05
0.5 (50%)	1.44
0.75 (75%)	1.99
0.9 (90%)	2.79

Problem
with non linear Probing: SECONDARY CLUSTERING

Two keys that hash to the same place follow the same collision path

Idea:

Double Hashing

Collision Resolution

Ex: **Open Adressing: (3) Double Hashing**

$$\underbrace{h(k_i)}_{h_0},\ \underbrace{h(k_i)+d(k_i)}_{h_1},\ \underbrace{h(k_i)+2d(k_i)}_{h_2},\ \underbrace{h(k_i)+3d(k_i)}_{h_3},\ \ldots$$

$$h_j(k_i) = [h(k_i) + j \cdot d(k_i)] \bmod N$$

OR

Ex:

$h(k_i)$, $h(k_i)+d(k_i)$, $h(k_i)+4\ d(k_i)$, $h(k_i)+9\ d(k_i)$, …

$$h_J(k_i) = [h(k_i) + j^2 \cdot d(k_i)] \bmod N$$

→ Choice of primary hashing function h( )
Choice of secondary hashing function d( )

$$h_j(k_i) = [h(k_i) + j \cdot d(k_i)] \bmod N$$

Example of Double Hashing

- $N = 13$
- $h(k) = k \bmod 13$
- $d(k) = 7 - k \bmod 7$

- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

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

$\Downarrow$

		41			18				22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

occupied

$$h_j(k_i) = [h(k_i) + j \cdot d(k_i)] \bmod N$$

Example of Double Hashing

- $N = 13$
- $h(k) = k \bmod 13$
- $d(k) = 7 - k \bmod 7$

• Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

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

⇩

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

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

Remove 22 (22 mod 13 = 9)

$\boldsymbol{h}(\boldsymbol{k})$ = $\boldsymbol{k}$ mod 13
$\boldsymbol{d}(\boldsymbol{k})$ = 7 - $\boldsymbol{k}$ mod 7

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

Search 31

Primary hash function: 31 mod 13 = 5 Occupied and different

Secondary hash function: 7 - 31 mod 7 = 4.
Probe cell 5+4=9: AVAILABLE

Probe cell (9+4) mod 13: FOUND

$$h(k) = k \bmod 13$$
$$d(k) = 7 - k \bmod 7$$

Another Example of Double Hashing

$$\boxed{\begin{array}{c} h(k_i) = k_i \bmod N \\ h'(k_i) = k_i \operatorname{div} N \end{array}}$$

N prime!

Performances of Double Hashing

Experimental results for a hash table with load factor α

Search

$\alpha = n/N$	$C(\alpha)$
0.1 (10%)	1.05
0.5 (50%)	1.38
0.75 (75%)	1.83
0.9 (90%)	2.55

Linear, Quadratic, Double hashing

α=n/N	$C(\alpha)$
0.1 (10%)	1.06
0.5 (50%)	1.50
0.75 (75%)	2.50
0.9 (90%)	5.50

α = n/N	$C(\alpha)$
0.1 (10%)	1.05
0.5 (50%)	1.44
0.75 (75%)	1.99
0.9 (90%)	2.79

α = n/N	$C(\alpha)$
0.1 (10%)	1.05
0.5 (50%)	1.38
0.75 (75%)	1.83
0.9 (90%)	2.55

Performance of Hashing: Summary

- In the worst case, searches, insertions and removals on a hash table take $O(n)$ time
- The worst case occurs when all the keys inserted into the dictionary collide
- The load factor $\alpha = n/N$ 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 approximately
  $$1 / (1 - \alpha)$$
- 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%
- Applications of hash tables:
  - small databases
  - compilers
  - browser caches
  - P2P