



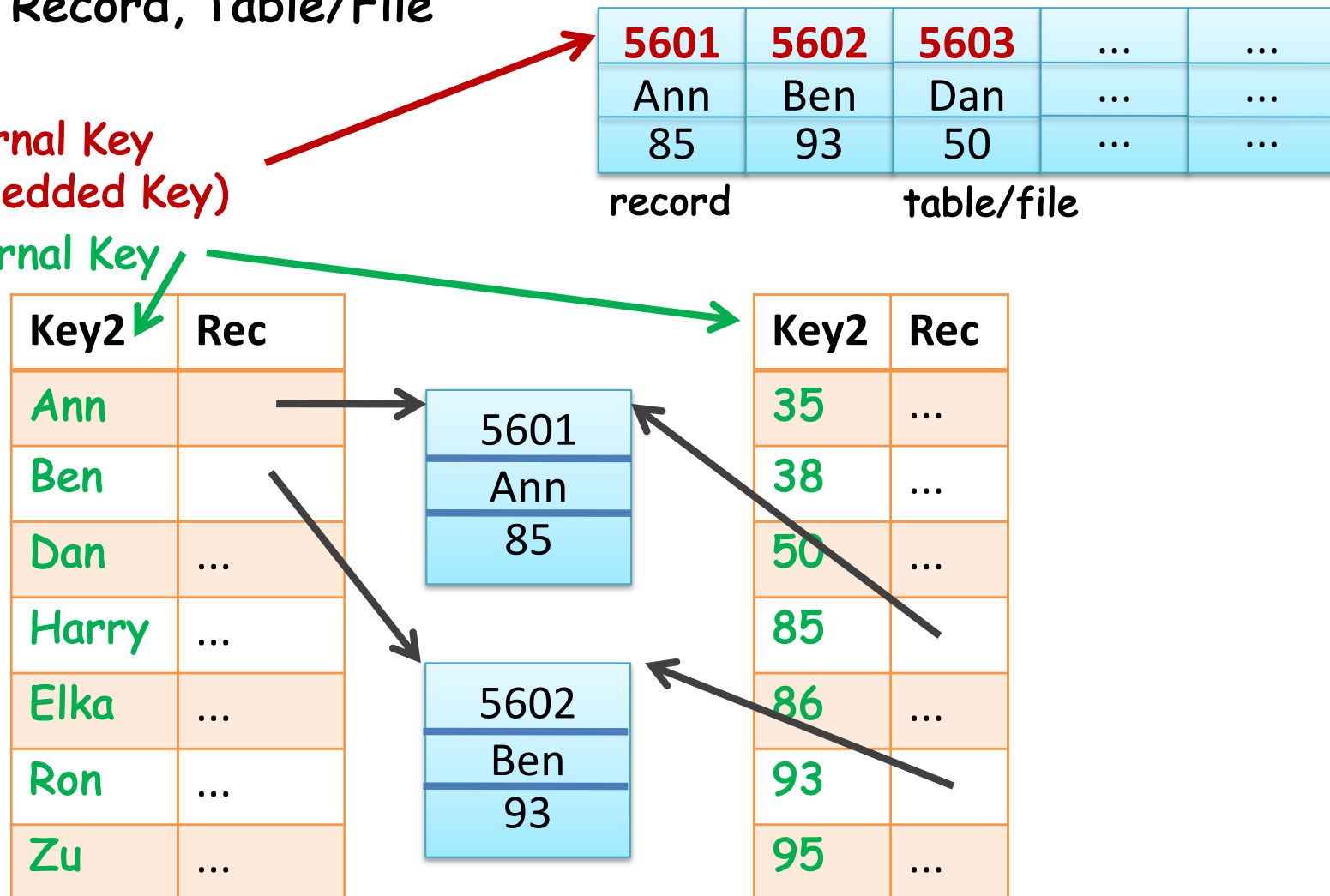
Searching

Kiatnarong Tongprasert

Searching Terminology

- Search, Record, Table/File
- Key

- Internal Key (Embedded Key)
- External Key



- Successful Search/Unsuccessful Search
- Internal Search (main memory) / External Search (auxiliary memory)

Searching

Search Unordered Table

- **Sentinel Search**
- **Move to Front**
- **Transposition**

Search Ordered Table

List (Array)	Tree Search
- Sequential Search (Linear Search)	- Sequential Search (Linear Search)
- Binary Search	- Binary Search Tree

- Hashing

Searching Unorder List

rec	19	56	2	7	25	18	...	40		
	1	2	3	4	5	6	...	n	pos	

```
found = false; //Typical version
i = 1;
loop (i<=n) and (not found)
  if (key == rec[i].key) {
    foundIndex = i;
    found = true;
  }
  else
    i = i + 1;
  end if
end loop
```

```
pos = n+1; //More efficient version
i = 1;
loop (i<>pos)
  if (key == rec[i].key)
    pos = i;
  else
    i = i+1;
  end if
endloop

if (i<=n)
  search = i;
else
  search = 0;
end if
```



Sentinel Search

rec	19	56	2	7	25	18	...	40	Sentinel	
	1	2	3	4	5	6	...	n		

```
rec[n+1] = key;    //adding sentinel
```

```
i = 1;
```

```
loop (key <> rec[i].key)
```

```
    i = i+1;
```

```
end loop
```

```
if (i < n)
```

```
    search = i;
```

```
else
```

```
    search = 0;
```

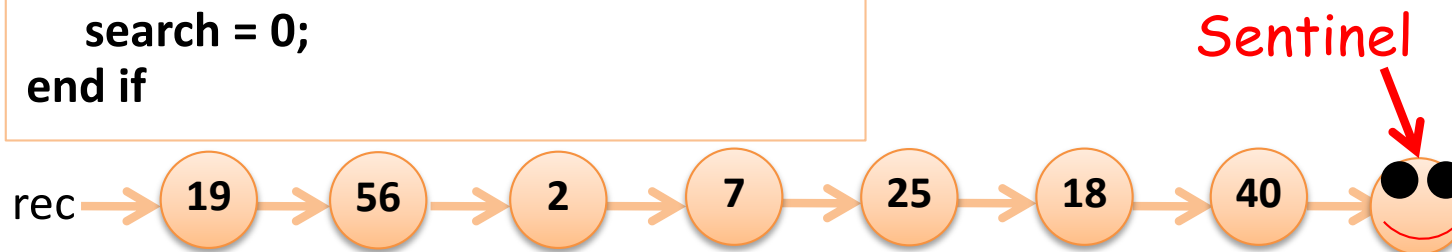
```
end if
```

$O(n)$

Best Case 1

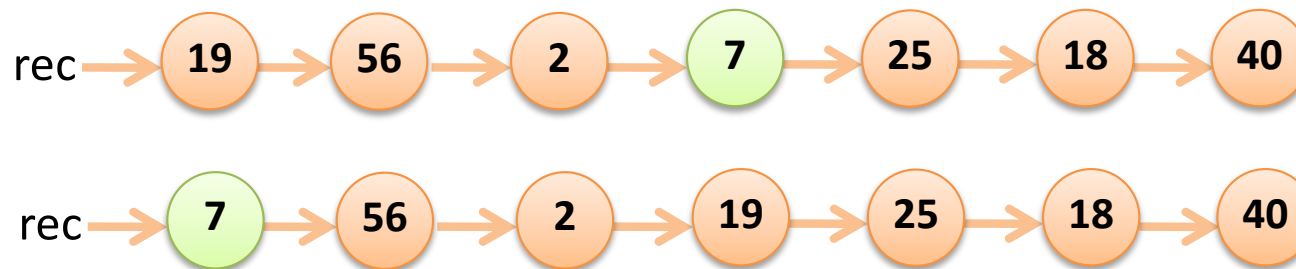
Worst Case n

Avg Case $(n+1)/2$



การใช้ **sentinel** ทำให้การทำงานเร็วขึ้น จากการที่ไม่ต้องตรวจสอบว่า **list** หมดแล้ว

Move to Front Heuristic



Move to Front

- แต่ละครั้งที่ search พบ ให้เลื่อน record นั้นขึ้นไปอยู่หน้าสุดของ list (ต้องเป็น linked list version)
- แนวคิด : ของอะไรที่ใช้แล้วมีแนวโน้มที่น่าจะถูกใช้อีกจึงเอามาไว้ข้างหน้า

A **heuristic** is a **mental shortcut** that allows people to **quickly make judgments** and solve problems. These mental shortcuts are **typically** informed **by our past experiences** and allow us to act quickly. However, heuristics are really more of a rule-of-thumb; **they don't always guarantee a correct solution.**



Transposition Heuristic

rec	19	56	2	7	25	18	...	40		
rec	19	56	7	2	25	18	...	40		

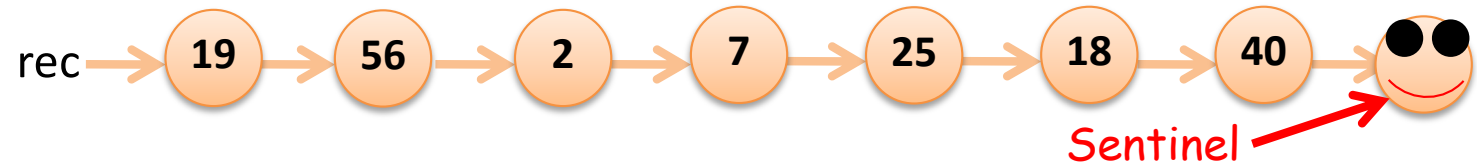
Transposition

- แต่ละครั้งที่ search พบ ให้สลับ record ที่ search พบขึ้นมาข้างหน้า 1 ตำแหน่ง
- แนวคิด : การใช้ครั้งเดียวไม่ได้แปลว่าจะใช้อีกครั้งหนึ่งเสมอไป แต่การสลับแบบนี้ หากใช้มากๆ ก็จะเลื่อนขึ้นมาอยู่ข้างหน้าเอง

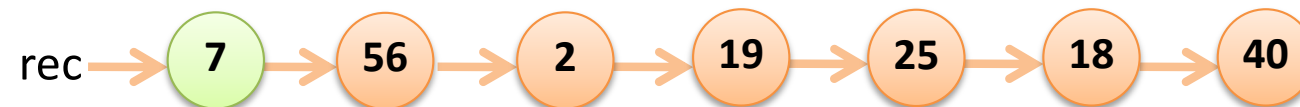
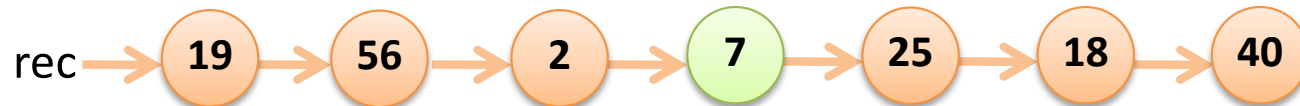
Searching **Unordered** Table

- **Sentinel Search**

rec	19	56	2	7	25	18	...	40	Sentinel	
-----	----	----	---	---	----	----	-----	----	----------	--



- **Move to Front**



- **Transposition**

rec	19	56	2	7	25	18	...	40		
-----	----	----	---	---	----	----	-----	----	--	--

rec	19	56	7	2	25	18	...	40		
-----	----	----	---	---	----	----	-----	----	--	--

Searching

Search Unordered Table

- Sentinel Search
- Move to Front
- Transposition

Search Ordered Table

List (Array)	Tree Search
- Sequential Search (Linear Search)	- Sequential Search (Linear Search)
- Binary Search	- Binary Search Tree

- Hashing

Sequential Search (Linear Search)

2	5	7	8	10	12	15	18	20	22
---	---	---	---	----	----	----	----	----	----



h



$O(n)$

Best Case 1

Worst Case n

Avg Case $(n+1)/2$

Searching

Search Unordered Table

- Sentinel Search
- Move to Front
- Transposition

Search Ordered Table

List (Array)	Tree Search
- Sequential Search (Linear Search)	- Sequential Search (Linear Search)
- Binary Search	- Binary Tree Search

- Hashing

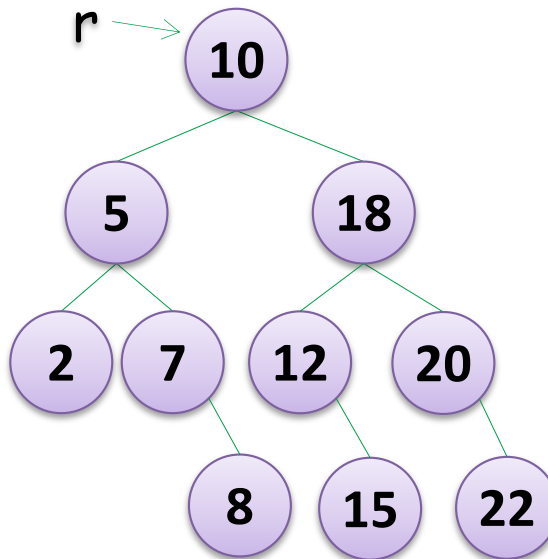
Binary (Tree) Search

	0	1	2	3	4	5	6	7	8	9
a	2	5	7	8	10	12	15	18	20	22

↑ ↑
2 1

binarySearch

```
low = 1, high = dataSize
loop ( low <= high )
    mid = ( low + high ) / 2
    if( a[ mid ] < key )
        low = mid + 1
    else if( a[ mid ] > key )
        high = mid - 1
    else low = high + 1
    end if
end loop
if (key == a[mid])
    return mid
else return -1
end if
```



$O(\log_2 n)$

binarySearch

```
p = q = root
loop ( p is not null )
    if( key < p(data) )
        p = p->left;
    else if ( key > p(data) )
        p = p->right;
    else
        q = p
        p = null
    end if
end loop
if (key == q->data)
    return q
else return null
end if
```

Hashing concepts



1. *searches* ก่อนๆ นี้ต้องเช็คเทียบหลายครั้งจึงพบหรือพบว่าไม่มี
การเช็คเทียบแต่ละครั้ง เรียก **probe**

2. ความต้องการของ hashing :

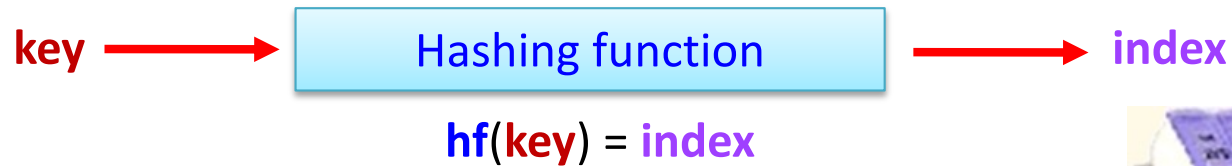
- ทำอย่างไรเราจะหาของได้ใน **1 probe** ?



เราต้องรู้ว่ามันอยู่ที่ไหน → เก็บอย่างมีหลักการ

Hasing Function

- ตัวตนของแต่ละ record อยู่ที่ไหน ?
อยู่ที่ **key** ของมัน ดังนั้นต้อง
โยง **key** เข้ากับ ที่เก็บ (array index)



- การโยง key ให้เป็นที่เก็บเรียก **hashing algorithm**
ฟังก์ชันที่ใช้เรียกว่า **hashing function**
เรา **hash key** ไปสู่ **index**
- Array ที่เก็บ records เรียก **hash table**.

Mapping Techniques

Subtraction

- ถ้าทุก id นำหน้าด้วย 54011

$$hf(key) = key - 54\ 01\ 1000$$

$$hf(54\ 01\ 1004) = 4$$

$$hf(54\ 01\ 1037) = 37$$

$$hf(54\ 01\ 1576) = 576$$

Extraction

เอาแค่บางส่วนของ key

$$54\ 01\ 1576 \rightarrow 4156$$

Summation, Division, ...

– Folding.

- แบ่ง key เป็นส่วนๆ
- summation(/subtraction/...) ส่วนนั้นๆ

$$54091576$$

$$= 540 + 915 + 76$$

$$= 1531$$

$$= 531 \quad // \text{array size} = 1000$$

- **Modulo** เพื่อให้อยู่ใน range

- **Midsquare**

key² หรือ part_of_key²

เอาตรงกลางมา

$$54011576 \rightarrow 576 * 576$$

$$= 331776 \rightarrow 177$$

Mapping String 1 - 2

Mapping String 1

string → sum all ASCII chars → int

HashString1

```
k = address of first char
HashVal = 0;
loop (*k != null)
    HashVal += *k++
endloop
return HashVal mod TableSize
```

- ปัญหา: table ใหญ่ 10,000 => กระจาย distribute ไม่ดี
- ASCII 0-127, 8 chars => $127 * 8 \Rightarrow [0-1,016]$

Mapping String 2

string → $(k[0] + 27 k[1] + 27^2 k[2]) \% \text{TableSize}$ → int

HashString2

```
k = address of first char
hv = (k[0]+27* k[1]+729* k[2]) % TableSize;
if (hv<0) // 27=26+blank
    return hv+= TableSize
else return hv
end if
```

ถ้าคิด 26 ตัวไม่นับ blank คิดเฉพาะ 3 chars แรก
=> 17,576 combinations => แต่จริง ๆ
แล้ว Eng. ไม่ random ตาม dic. ได้ 2,851
combinations = 28% (ของ 10,000) ดังนั้น
table ใหญ่ ใช้ไม่เต็ม กระจายไม่ดี

Mapping String 3

Horner's rule : Polynomial of 32

string →

$$(32^4 k[0] + 32^3 k[1] + 32^2 k[2] + 32 k[3] + k[4]) \% \text{ Tablesize}$$

→ int

Horner's rule : Polynomial of 32

k = address of first char

i = keysize-1

loop (*k != null)

 hv += 32**i + *k++

 i -= 1

end loop

hv = hv % Tablesize

return hv

$$\sum_{i=0}^{\text{keysize}-1}$$

$$32^{\text{keysize}-1-i} * k[i]$$

$$\sum_{i=0}^{\text{keysize}-1}$$

$$k[\text{keysize} - 1 - i] + 32^i$$

- if keysize = 5
- iteration #0 : $32^0 * K[0]$
- iteration #1 : $32^1 * K[0] + 32^0 * K[1]$
- iteration #2 : $32^2 * K[0] + 32^1 * K[1] + 32^0 * K[2]$
- iteration #3 : $32^3 * K[0] + 32^2 * K[1] + 32^1 * K[2] + 32^0 * K[3]$
- iteration #4 : $32^4 * K[0] + 32^3 * K[1] + 32^2 * K[2] + 32^1 * K[3] + 32^0 * K[4]$

Mapping String 3
Horner's rule : Polynomial of 32

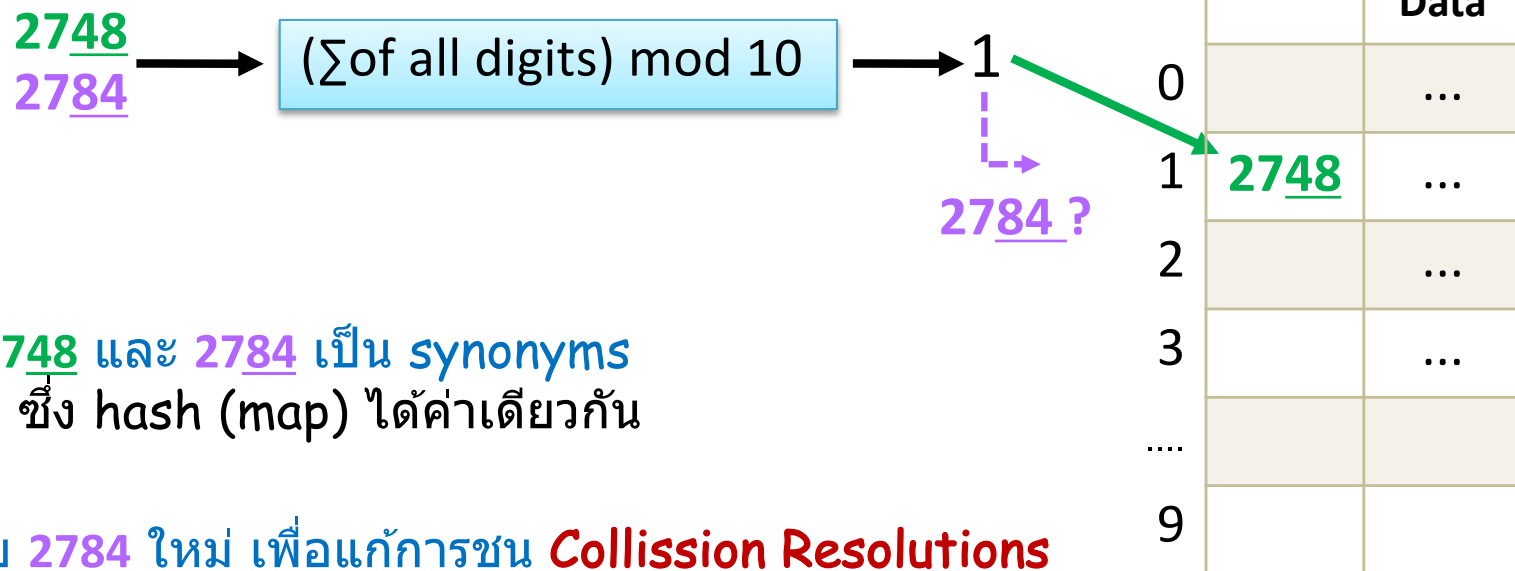
Mapping String

- `string` ยาวๆ ใช้เวลามาก ==> `truncation`
- 438 Washington NY
= 438WaNY
map -> array range

Collision, Synonym

Collision เกิดเมื่อ hash แล้วได้ index ที่มีของอยู่แล้ว เช่น

- insert **2748** $\text{hash}(2748) = 1$ วาง ใส่ใน slot 1
- ,, **2784** $\text{hash}(2784) = 1$ ไม่ว่าง collision ชน กับ 2748 ต้องหาที่เก็บใหม่ให้



Synonyms : **2748** และ **2784** เป็น synonyms
set of keys ซึ่ง hash (map) ได้ค่าเดียวกัน

- ต้องหาที่เก็บ **2784** ใหม่ เพื่อแก้การชน **Collision Resolutions**
 - **Open Addressing (Closed Hashing)**
ที่เก็บใหม่อยู่ใน table เดิม ซึ่งแก้การชนไม่ได้ 100%
 - **Separate Chaining** ที่เก็บใหม่นอก table เดิม แก้การชนได้ 100%

Collision Resolutions

2748
2784
7284

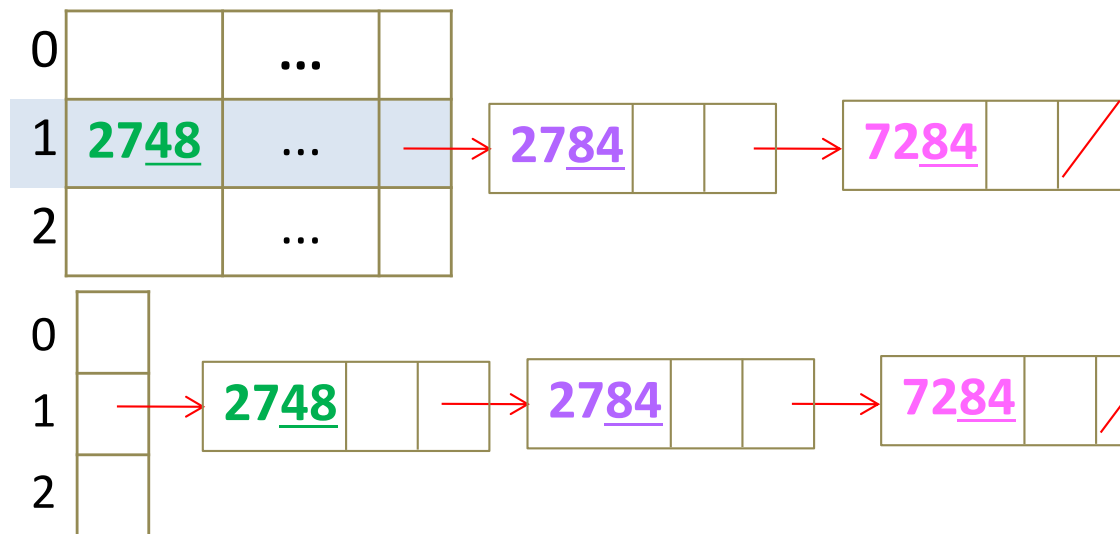
(\sum of all digits) mod 10

0		...
1	2748	...
2	2784	...
3	7284	...

Open Addressing (Closed Hashing)

ที่เก็บใหม่อยู่ใน table เดิม ซึ่งแก้การชนไม่ได้ 100%
เช่น 2784 ชนกับ 2748 จึงหาที่ใหม่ให้ ในตัวอย่าง
- ให้อยู่ถัดมา 1 ช่อง วิธีนี้เรียกว่า **Linear Probing**

Separate Chaining ที่เก็บใหม่อยู่นอก table เดิม แก้การชนได้ 100%



นอกจาก sorted(นิยม) linked list
อาจใช้โครงสร้างอื่นได้ เช่น
binary search tree
hash table

Good Hashing Function, Load Factor

Hashing Function ที่ดี

- อยู่ใน index range
==> mod tablesize
==> table size ควรเป็น prime.
- คำนวณ ง่าย รวดเร็ว
- กระจายดี กระจายทั่วถึง
- collision น้อย
- การคำนวณที่ใช้ทั้ง key แทนได้ดีกว่าที่ใช้บางส่วน

Collision การชนกัน

Collision (การชน) เกิดขึ้นเมื่อ hash ได้ index ที่มีของอยู่แล้ว จึงต้องหาที่เก็บใหม่ให้
มีทางแก้ 2 วิธี

1. Separate Chaining

ที่เก็บใหม่อยู่นอก table เดิม แก้การชนได้ 100%

2. Open Addressing

ที่เก็บใหม่อยู่ใน table เดิม ซึ่งแก้การชนไม่ได้ 100%

Separate Chaining

ที่เก็บใหม่ **อยู่นอก table เดิม** แก้การชนได้ 100%

Separate Chaining (cont.)

- **Load factor λ** = ปริมาณข้อมูล ใน hash table / ขนาดของตาราง
 - ในรูปตัวอย่าง $\lambda = 10/10 = 1.0$
 - จากการวิเคราะห์ : table size ไม่ค่อยสำคัญเท่าไร ที่สำคัญคือ λ
 - general rule สำหรับ separate chaining hash table
 - $\lambda \sim 1$
 - table size เป็น prime
 - ข้อเสียของ separate chaining คือ ใช้ data structure ที่ 2 เช่น linked list ซึ่งทำให้ช้าลง
(allocate new node, implement 2nd data structure)
 - ข้อดี : 100% solved collision
- $h(x) = x \bmod 10$
 - Table size = 10
(not prime for simplicity)

0	
1	--->81 --->1
2	
3	
4	--->64 --->4
5	--->25 --->55
6	--->36 --->16
7	
8	
9	--->49 --->9

Open Addressing
ที่เก็บใหม่อยู่ใน table เดิม ซึ่งแก้การชนไม่ได้ 100%

Linear Probing

probe ครั้งที่ i ลอง $h(k) + f(i)$

Linear Probing $f(i) = i$ ลอง : $h(k), h(k)+1, h(k)+2, h(k)+3, \dots$

Quadratic Probing $f(i) = i^2$ ลอง : $h(k), h(k)+1, h(k)+4, h(k)+9, \dots$

Rehash: hash อีกครั้งเพื่อให้ได้ที่ใหม่

$$hf(key) = key \bmod 10$$

$$h(1111) = 1$$

$$\text{rehash ครั้งที่ } 1 = 1 + 1 = 2 \text{ ชน}$$

$$\text{rehash ครั้งที่ } 2 = 1 + 2 = 3 \text{ ชน}$$

$$\text{rehash ครั้งที่ } 3 = 1 + 3 = 4 \text{ วางใส่ได้}$$

รวมทั้งหมด 4 probes

0		...
1	2151	...
2	2872	...
3	4553	...
4	1111	
...
9		

array size = 10

(not prime for simplicity)

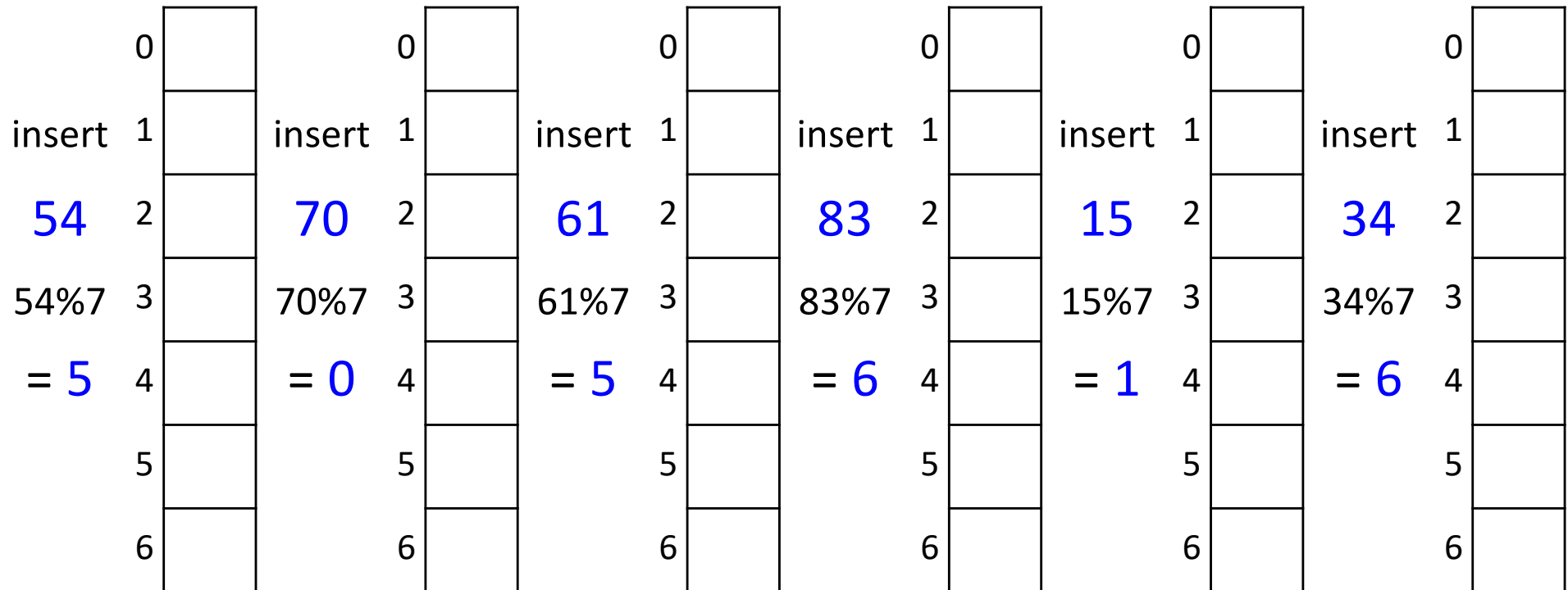
Linear Probing

- การคำนวณง่าย search ง่าย
- มีแนวโน้มให้เกิดการกระจุกตัว (**Clustering**) ของ data
→ collision

Linear Probing

probe ครั้งที่ i ลอง $h(k) + f(i)$

Linear Probing $f(i) = i$ ลอง : $h(k), h(k)+1, h(k)+2, h(k)+3, \dots$



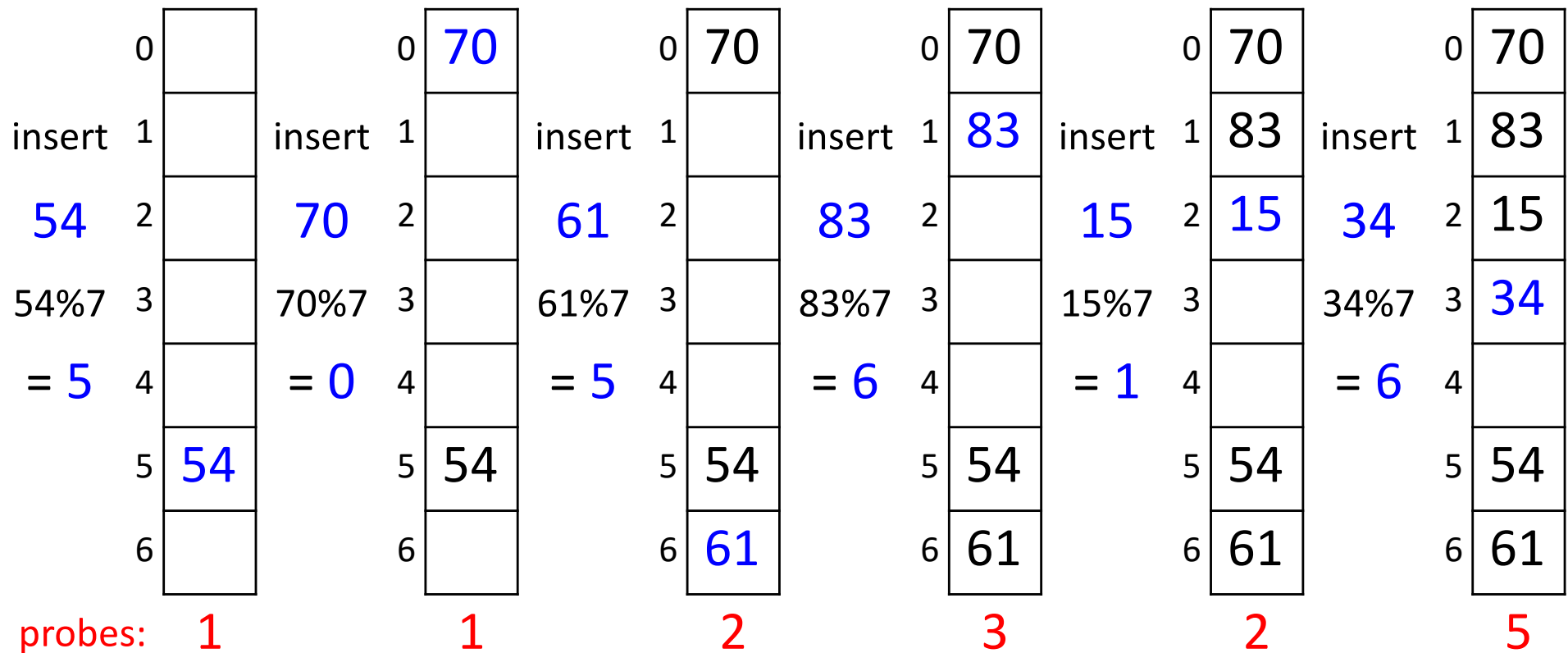
probes:

$$h(\text{key}) = \text{key} \bmod 7$$

Linear Probing

probe ครั้งที่ i ลอง $h(k) + f(i)$

Linear Probing $f(i) = i$ ลอง : $h(k), h(k)+1, h(k)+2, h(k)+3, \dots$



$$h(\text{key}) = \text{key} \bmod 7$$

Quadratic Probing

probe ครั้งที่ i ลอง $h(k) + f(i)$

Quadratic Probing $f(i) = i^2$ ลอง : $h(k)$, $h(k)+1$, $h(k)+4$, $h(k)+9$,...

Linear Probing $f(i) = i$ ลอง : $h(k)$, $h(k)+1$, $h(k)+2$, $h(k)+3$,...

1111

$hf(key)$
= $key \bmod 10$

$$1+9 = 10 \bmod 10$$

$$= 0 \text{ ว่าง}$$

$$1 \text{ --- ชน}$$

$$1+1 = 2 \text{ ชน}$$

$$1+4 = 5 \text{ ชน}$$

ทั้งหมด 4 probes

0	1111	...
1	2151	...
2	2872	...
3		...
4	1544	
5	1115	
...
9		

array size = 10

(not prime for simplicity)

Quadratic Probing

- มีการกระจาย(distributes) มากกว่า Linear Probing
- ปัญหาของ quadratic probing คือ เราไม่ได้ probe ทุกช่องของ table แต่สามารถพิสูจน์ได้ว่า หาก table size เป็น prime และ table ว่างอย่างน้อย $\frac{1}{2}$ quadratic probing จะสามารถหาช่องว่างได้ และ ไม่มีการ probe ซ้ำช่องเดิม
- แต่การที่ table ต้องว่างอย่างน้อย $\frac{1}{2}$ ทำให้ quadratic probing : space inefficient

Quadratic Probing

probe ครั้งที่ i ลอง $h(k) + f(i)$

Quadratic Probing $f(i) = i^2$ ลอง : $h(k), h(k)+1, h(k)+4, h(k)+9, \dots$

0		0		0		0		0		0	
insert	1	insert	1	insert	1	insert	1	insert	1	insert	1
54	2	72	2	61	2	1	2	33	2	70	2
54%7	3	72%7	3	61%7	3	1%7	3	33%7	3	70%7	3
= 5	4	= 2	4	= 5	4	= 1	4	= 5	4	= 0	4
5		5		5		5		5		5	
6		6		6		6		6		6	

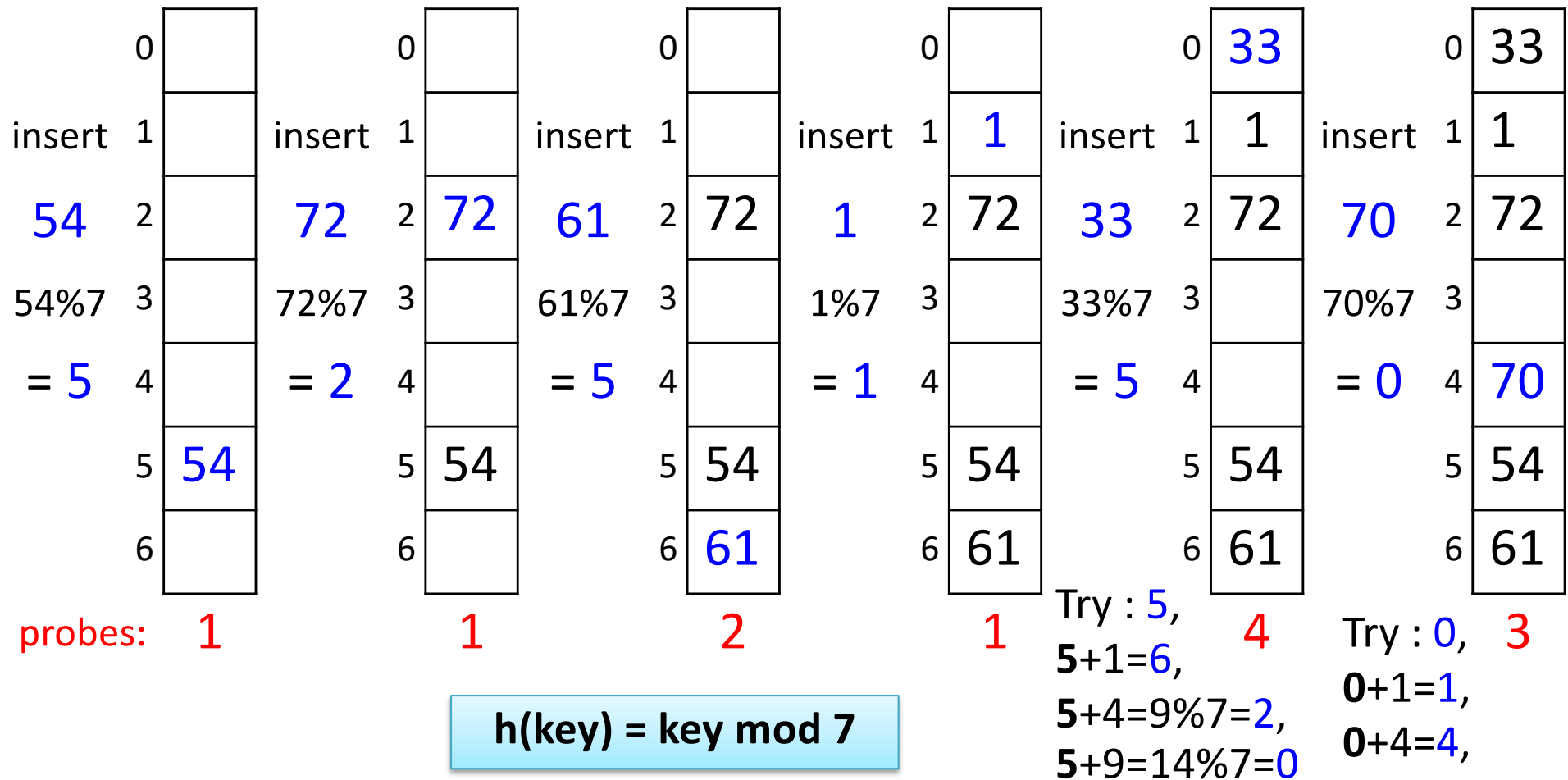
probes:

$$h(\text{key}) = \text{key} \bmod 7$$

Quadratic Probing

probe ครั้งที่ i ลอง $h(k) + f(i)$

Quadratic Probing $f(i) = i^2$ ลอง : $h(k)$, $h(k)+1$, $h(k)+4$, $h(k)+9$,...



Rehashing

Rehashing

- ถ้า table แน่นเกินไปจะเริ่มทำให้แต่ละ operation ใช้เวลานาน และอาจ insert ไม่ได้สำหรับ open addressing แบบ quadratic
- แก้ได้โดย สร้าง table ขึ้นใหม่ ใหญ่ขึ้น 2 เท่า และนำ data จาก table เดิมทั้งหมด มา hash ใส่ table ใหม่ เรียกว่า rehashing (rehash ทุก data เข้า table ใหม่) (rehash อีกความหมายหนึ่งคือ hash อีกครั้งเมื่อ collision)
- เมื่อใดบ้างที่ต้อง rehash
 - ทันทีที่ table เต็ม
 - เมื่อ insert ไม่ได้ (นโยบายเข้มงวด)
 - table เต็มถึงระดับที่กำหนดไว้ (ถึงค่า load factor ที่กำหนด) (นโยบาย เข้มงวดปานกลาง)

Rehashing

0	6
1	15
2	
3	24
4	
5	
6	13

0	6
1	15
2	23
3	24
4	
5	
6	13

Rehashing

- สร้าง table ขึ้นใหม่ ขนาด 17
(ค่า prime ถัดไปที่ใหญ่ขึ้น 2 เท่า)
- insert data ใหม่ทั้งหมด ด้วย
 $h(x) = x \bmod 17$
- แพง = $O(n)$

Insert :
13, 15, 6, 24

Insert : 23
Over 70 % full

Rehashing

$h(x) = x \bmod 7$, Linear Probing

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

ข้อเสียของการเก็บข้อมูลแบบ hash

- การจัดการที่เกี่ยวกับอันดับของข้อมูล

- หาค่าตัวน้อยสุด
- หาค่าตัวมากที่สุด
- ตัวที่ถัดจากตัวที่กำหนดให้

เนื่องจากการต้องดูข้อมูลทุกตัวทุกตำแหน่งในตารางจึงจะตอบได้ และจากการที่ ฟังก์ชันแฮชทำให้คีย์กระจาย ข้อมูลที่อยู่ใกล้กันเมื่อผ่านฟังก์ชันแฮชก็ไม่สามารถทราบได้ว่าอยู่ที่ใด

- ต้องระวังเรื่องฟังก์ชันแฮชที่ได้ค่าเหมือนกันทุกครั้ง

- ในกรณีที่นำคีย์ผ่านฟังก์ชันแฮชแล้วได้ค่าซ้ำเดิม ซึ่งระบบจะแก้ปัญหาการชนไว้แล้ว แต่หากเพิ่มข้อมูลเข้าไปมาก ๆ ก็จะทำให้ระบบเริ่มช้าลงได้ จึงต้องระวังเรื่องการใช้งานฟังก์ชันแฮช

สรุป

- การค้น การเพิ่ม การลบ ข้อมูลในตารางแฮชทำได้เร็ว
- สามารถปรับเวลาการทำงานให้เร็วขึ้น ด้วยการใช้น้ำที่เพิ่มขึ้น เพื่อให้ได้ λ ที่เหมาะสม
- ฟังก์ชันแฮชมีผลต่อประสิทธิภาพการทำงาน

Done