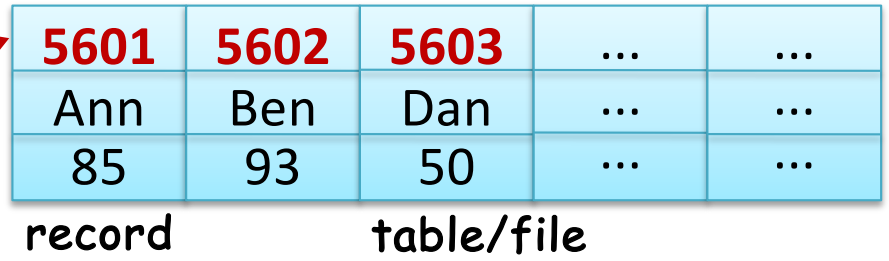
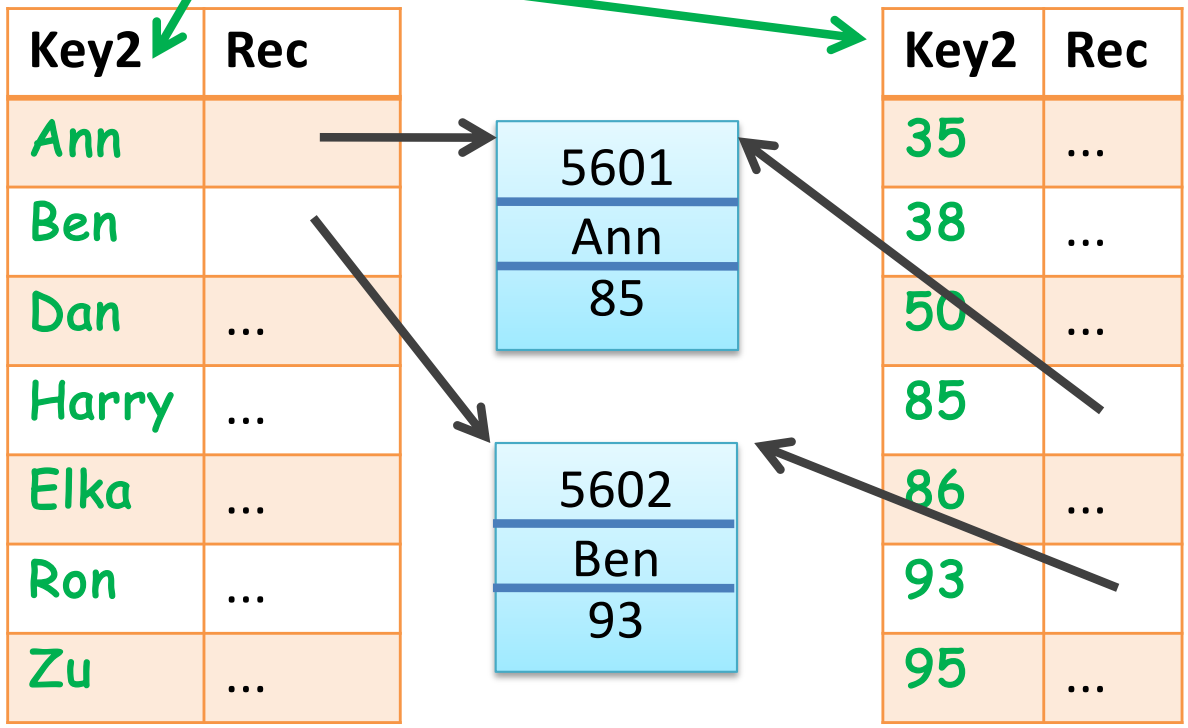

Searching
Kiatnarong Tongprasert

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

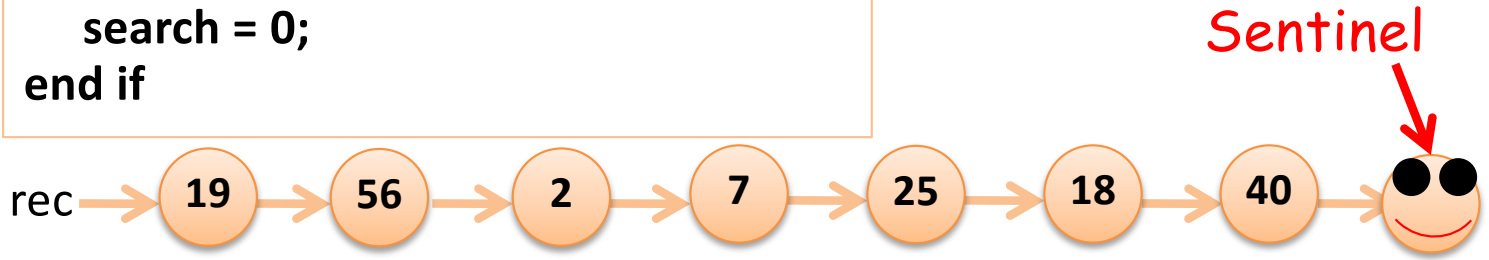
rec	19	56	2	7	25	18	...	40	Sentinel	
	1	2	3	4	5	6	...	n		

```
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
```

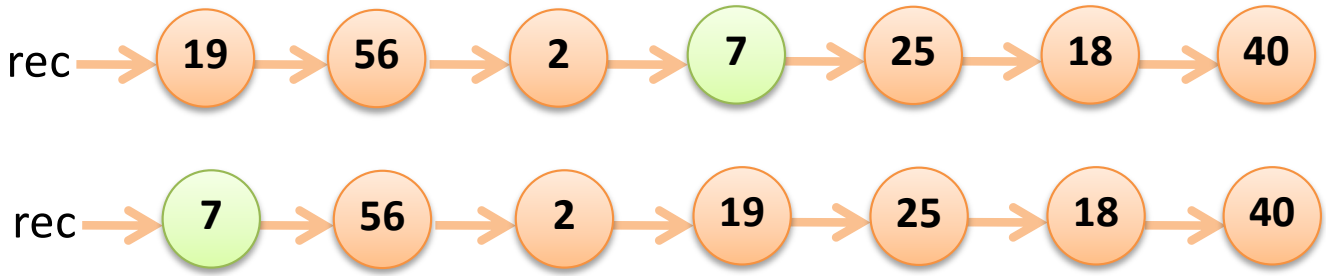
O(n)

Best Case	1
Worst Case	n
Avg Case	(n+1)/2



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

Move to Front Heuristic

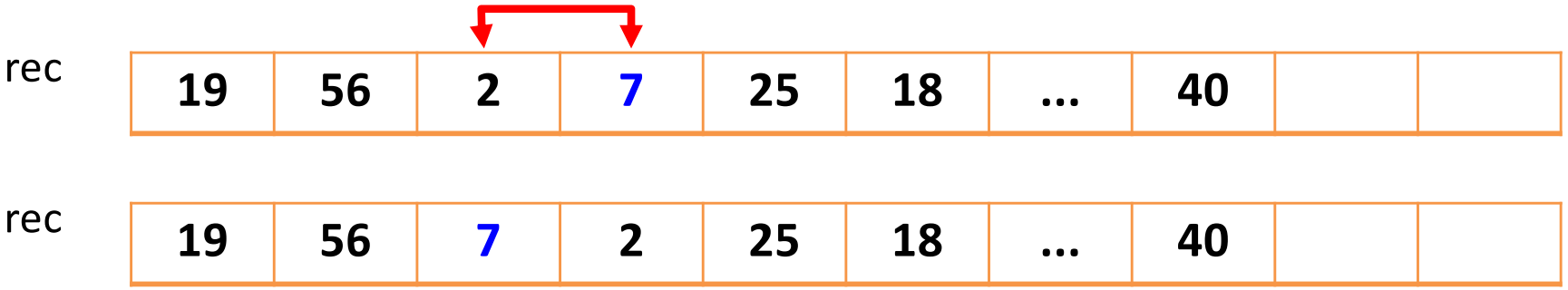


Move to Front

- 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

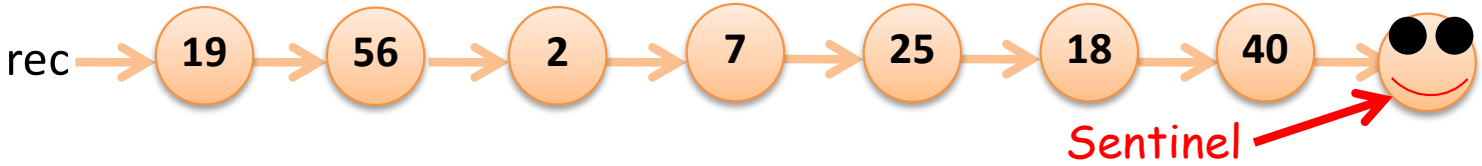


Transposition

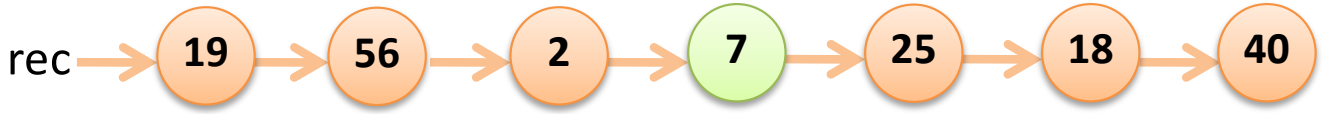
- แต่ละครั้งที่ search พบ ให้สลับ record ที่ search พบขึ้นมาข้างหน้า 1 ตำแหน่ง
- แนวคิด : การใช้ครั้งเดียวไม่ได้แปลว่าจะใช้อีกครั้งหนึ่งเสมอไป แต่การสลับแบบนี้หากใช้มากๆ ก็จะเลื่อนขึ้นมาอยู่ข้างหน้าเอง

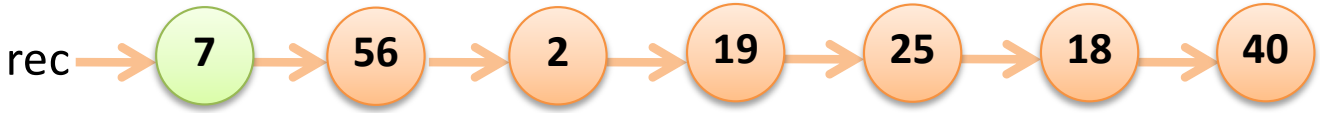
Searching Unordered Table

- **Sentinel Search**



- **Move to Front**





- **Transposition**

Searching

Search Unordered Table

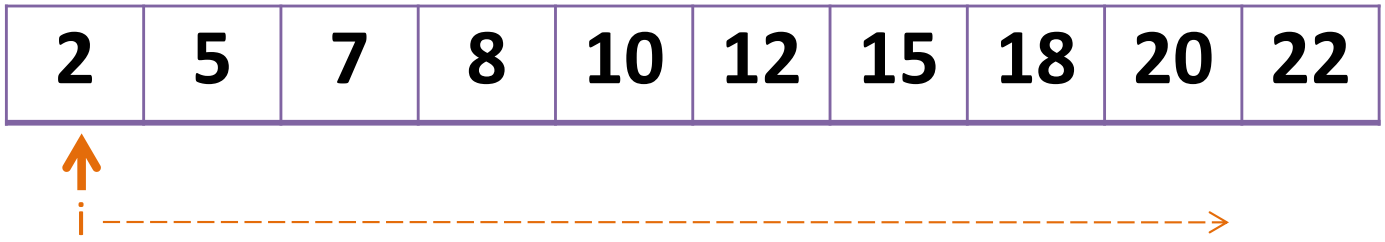
- Sentinel Search
- Move to Front
- Transposition

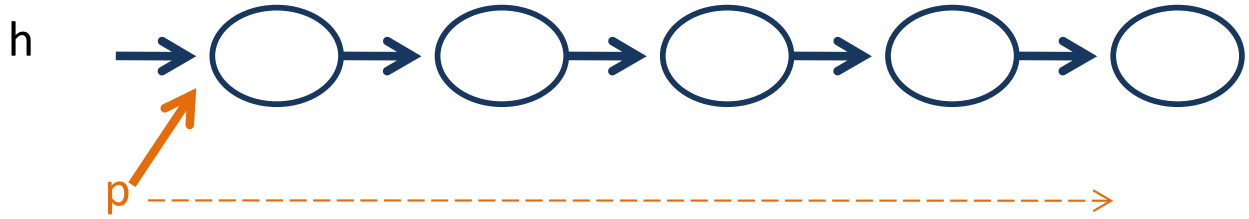
Search Ordered Table

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

- Hashing

Sequential Search (Linear Search)





O(n)

Best Case	1
Worst Case	n
Avg Case	(n+1)/2

Searching

Search Unordered Table

- Sentinel Search
- Move to Front
- Transposition

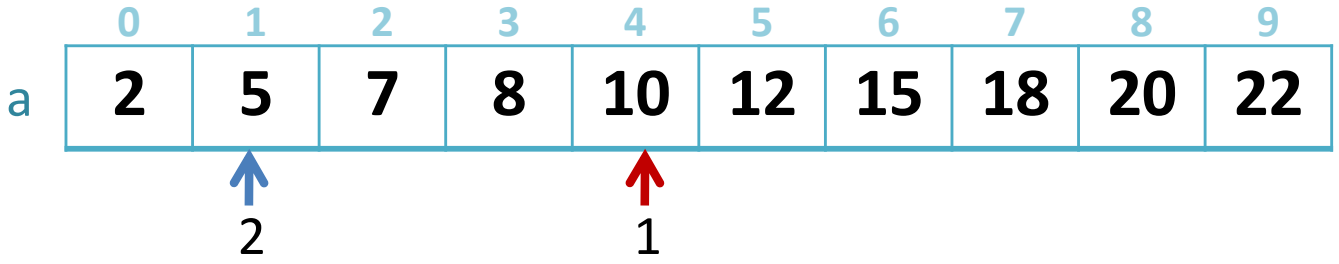
Search Ordered Table

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

- Hashing

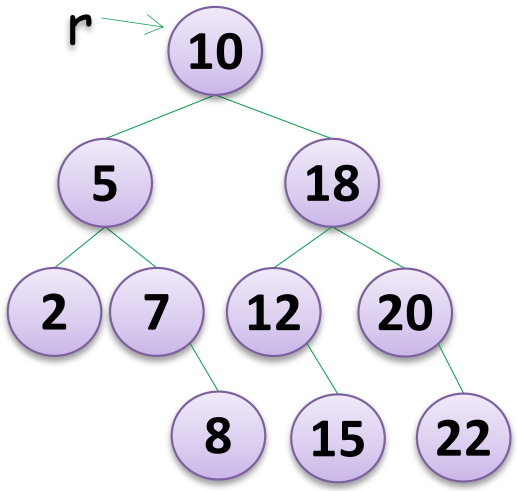
Binary (Tree) Search

Complexity: **O(log n)**
Best case: **1**
Worst case: **log n**
Avg: **log n**




```
binarySearch
    low = 1, high = dataSize
    while ( low <= high) {
      mid = ( low + high ) / 2 // คำนวณตำแหน่งกลาง (calculate middle index)
      if( key > a[mid] ) {        // ถ้าคีย์ใหญ่กว่า ⟶ ไปค้นหาในฝั่งขวา
            low = mid + 1
       }
      else if( key < a[mid] ) {   // ถ้าคีย์เล็กกว่า ⟶ ไปค้นหาในฝั่งซ้าย
            high = mid – 1
      }
      else
            return mid            // เจอแล้ว ⟶ คืนค่าตำแหน่ง
      end if
    }
    return  -1                    // ไม่พบข้อมูล
```

Binary (Tree) Search



Complexity: **O(log n)**
Best case: **1**
Worst case: **log n**
Avg: **log n**

binarySearch

```
p = q = root     // เริ่มต้นจากโหนดราก (start from root)
while ( p is not null) {
   if( key < p(data)) {
         p = p->left        // ถ้าคีย์เล็กกว่า ⟶ ไปทางซ้าย
   }
    else if ( key > p(data)) {
         p = p->right       // ถ้าคีย์ใหญ่กว่า ⟶ ไปทางขวา
    }
    else
         return  p          // เจอแล้ว ⟶ คืนค่าโหนดที่เจอ
  }
  return null               // ไม่พบคีย์
```

Hashing concepts

1. searches ก่อนๆ นี้ต้องเช็คเทียบหลายครั้งจึงพบหรือพบว่าไม่มี การเช็คเทียบแต่ละครั้ง เรียก probe
2. ความต้องการของ hashing :
   - ทำอย่างไรเราจะหาของได้ใน 1 probe ?

→ เราต้องรู้ว่ามันอยู่ที่ไหน → เก็บอย่างมีหลักการณ์

Hasing Function

- ตัวตนของแต่ละ record อยู่ที่ไหน ?

  อยู่ที่ key ของมัน ดังนั้นต้อง

  โยง key เข้ากับ ที่เก็บ (array index)

hf(key) = index

- การโยง key ให้เป็นที่เก็บเรียก hashing algorithm

  ฟังก์ชั่นที่ใช้เรียกว่า hashing function

  เรา hash key ไปสู่ index

- Array ที่เก็บ records เรียก hash table.

Mapping Techniques

Subtraction

- ถ้าทุก id นำหน้าด้วย 54 01 1

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 -> 4156

Summation, Division, ...

– **Folding.**

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

  54091576

  = 540 +915+76

  = 1531

  = 531 //array size = 1000

- **Modulo** เพื่อให้อยู่ใน range
- **Midsquare**

  key^2 หรือ $part\_of\_key^2$ เอาตรงกลางมา

  54011576 -> 576*576

  = 331776 -> 177

Mapping String 1 - 2

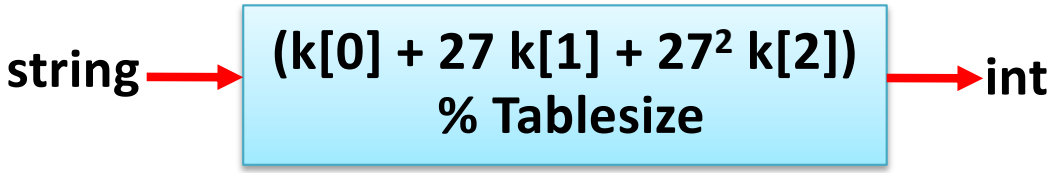
Mapping String 1

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 =>[0-1,016]

Mapping String 2



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 ดูตาม dic. ได้ 2,851 combinations = 28% (ของ 10,000) ดังนั้น table ใหญ่ ใช้ไม่เต็ม กระจายไม่ดี

Mapping String 3
Horner's rule : Polynomial of 32

string → $(32^4 k[0]+32^3 k[1]+32^2 k[2]+32k[3]+k[4])$ % Tablesize → int

$$\sum_{i=0}^{keysize-1} k[i] * 32^{keysize-1-i}$$

Horner's rule : Polynomial of 32

```
hv = 0
for (i = 0; i< keysize; i++) :
        hv =  hv * 32 + k[i]
hv = hv % Tablesize
```

- if keysize = 5 , hv = 0
- iteration #0 : hv = 0*32 + k[0]
- iteration #1 : hv = hv*32 + k[1]
- iteration #2 : hv = hv*32 + k[2]
- iteration #3 : hv = hv*32 + k[3]
- iteration #4 : hv = hv*32 + k[4]

สมมติ key = "HELLO", Tablesize = 101

•ค่าตัวอักษร (ASCII): H=72, E=69, L=76, O=79

การคำนวณ:

1.hv = 0

2.hv = 0*32 + 72 = 72

3.hv = 72*32 + 69 = 2373

4.hv = 2373*32 + 76 = 76012

5.hv = 76012*32 + 76 = 2432440

6.hv = 2432440*32 + 79 = 77838159

7.hv % 101 = 77838159 % 101 = 22

ผลลัพธ์: **hash("HELLO") = 22**

Mapping String

- string ยาวๆ ใช้เวลามาก ==> **truncation**
- 438 Washington NY

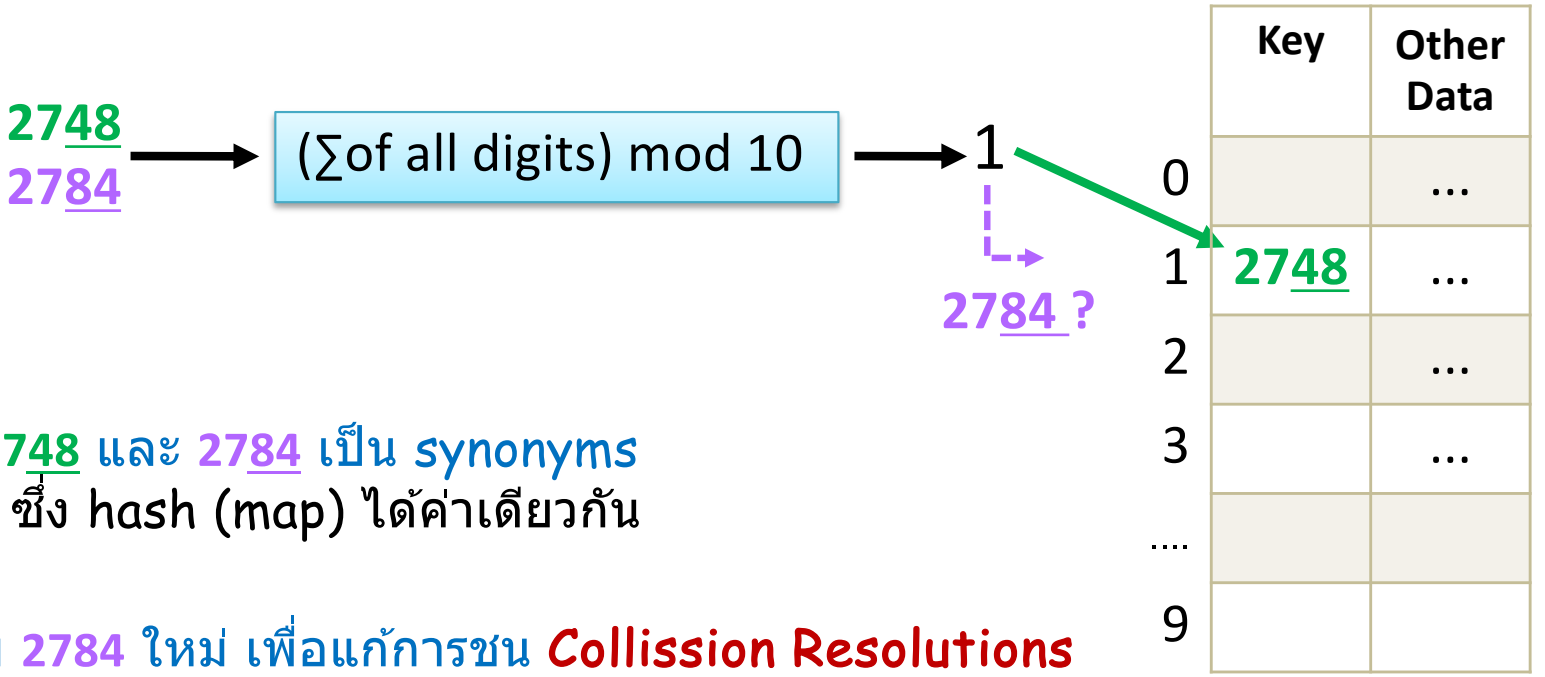
  = 438WaNY

  map -> array range

Collission, Synonym

Collission เกิดเมื่อ hash แล้วได้ index ที่มีของอยู่แล้ว เช่น

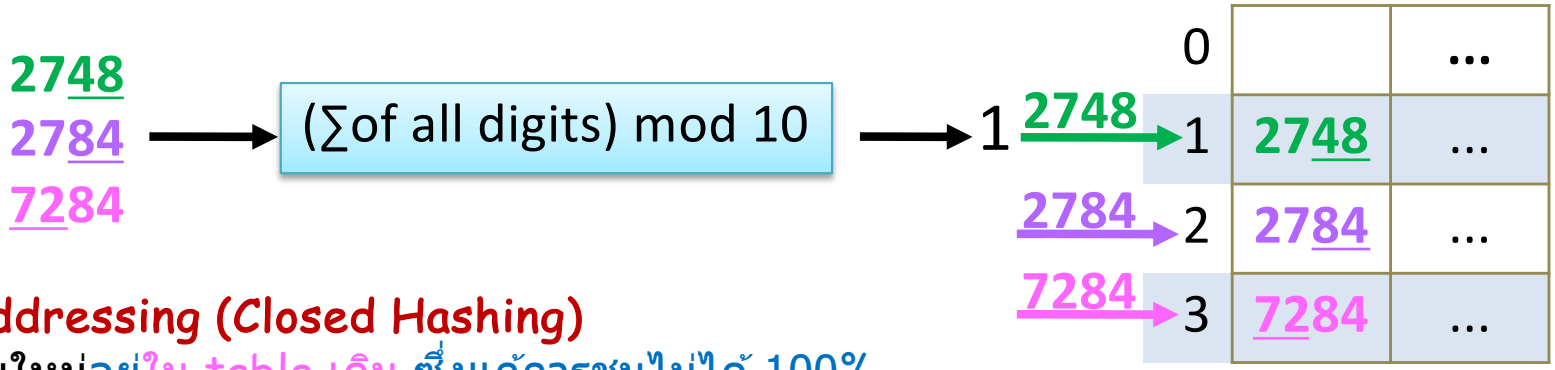
- insert 2748 hash(2748) = 1 ว่าง ใส่ใน slot 1
- ,, 2784 hash(2784) = 1 ไม่ว่าง collission ชน กับ 2748 ต้องหาที่เก็บใหม่ให้



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

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

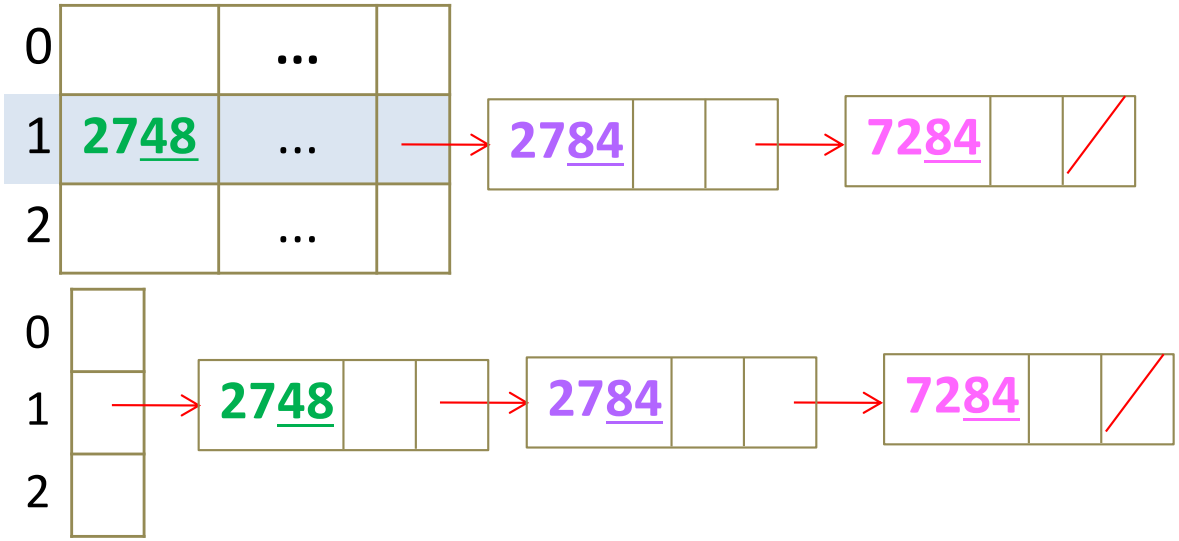
Collission Resolutions



Open Addressing (Closed Hashing)

ที่เก็บใหม่อยู่ใน table เดิม ซึ่งแก้การชนไม่ได้ 100%
เช่น2784 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%

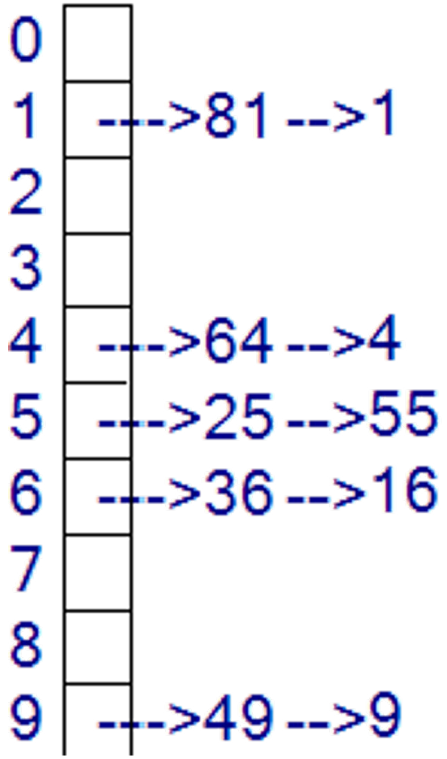
Separate Chaining

ที่เก็บใหม่อยู่นอก table เดิม แก้การชนได้ 100%

Separate Chaining

- **Load factor λ** = ปริมาณข้อมูล ใน hash table / ขนาดของตาราง
  - ในรูปตัวอย่าง λ = 10/10 = 1.0
- table size ไม่ค่อยสำคัญเท่าไหร่ ที่สำคัญคือ λ
- General rule สำหรับ separate chaining hash table
  - $\lambda \sim 1$
  - table size เป็น prime

- h(x) = x mod 10
- Table size = 10 (not prime for simplicity)



Separate Chaining (cont.)

- ข้อเสีย
  - ใช้ data structure ที่ 2 เช่น linked list
  - ทำให้การจัดเก็บ และ ค้นหา ช้าลง
- ข้อดี :
  - 100% solved collission

Open Addressing

ที่เก็บใหม่อยู่ใน table เดิม ซึ่งแก้การชนไม่ได้ 100%

Linear Probing

probe ชนครั้งที่ i	ลอง h(k) + f(i)
Linear Probing f(i) = i	ลอง : h(k), h(k)+1, h(k)+2, h(k)+3,...
Quadratic Probing f(i) = i^2	ลอง : h(k), h(k)+1, h(k)+4, h(k)+9,...

Rehash: hash อีกครั้งเพื่อให้ได้ที่ใหม่

hf(key) = key mod 10

h(1111) = 1 ชน

rehash ครั้งที่ 1 = 1+1 = 2 ชน

rehash ครั้งที่ 2 = 1+2 = 3 ชน

rehash ครั้งที่ 3 = 1+3 = 4 ว่าง ใส่ได้

รวมทั้งหมด 4 probes

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

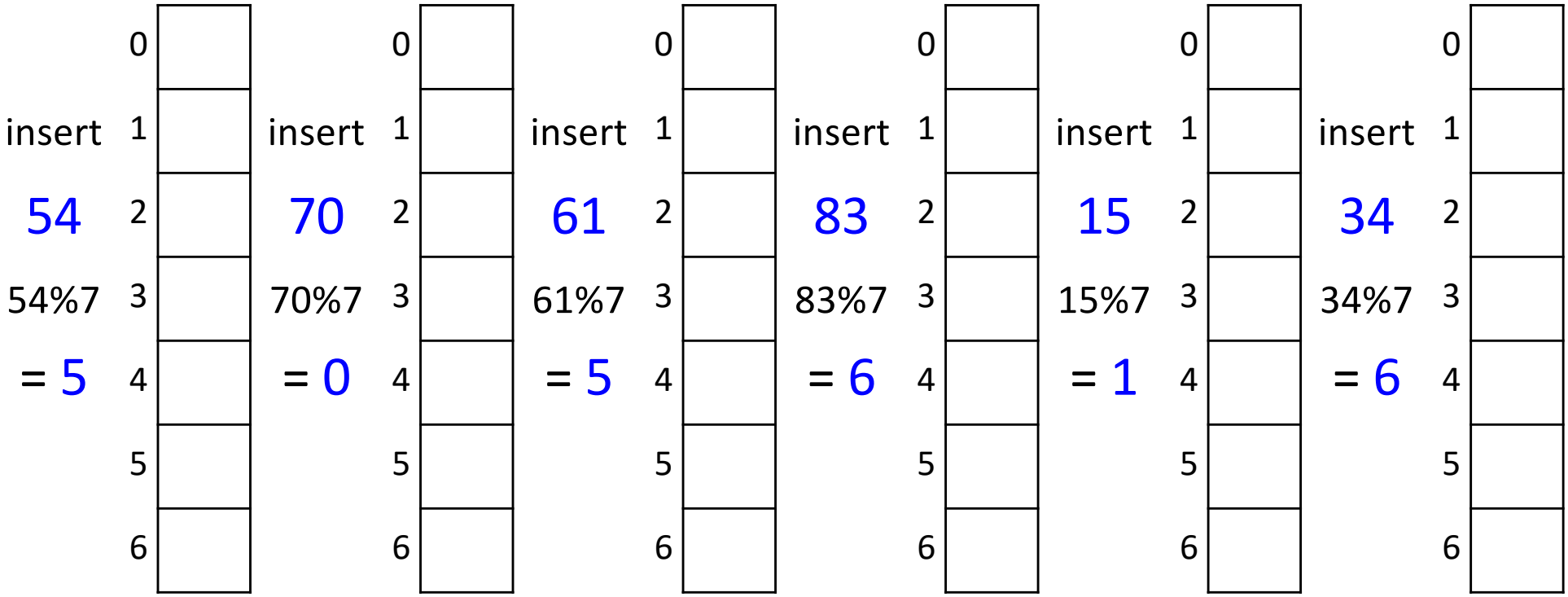
array size = 10
(not prime for simplicity)

Linear Probing

- การคำนวณง่าย search ง่าย
- มีแนวโน้มให้เกิด
  การกระจุกตัว (**Clustering**) ของ data
  -> collission

Linear Probing

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

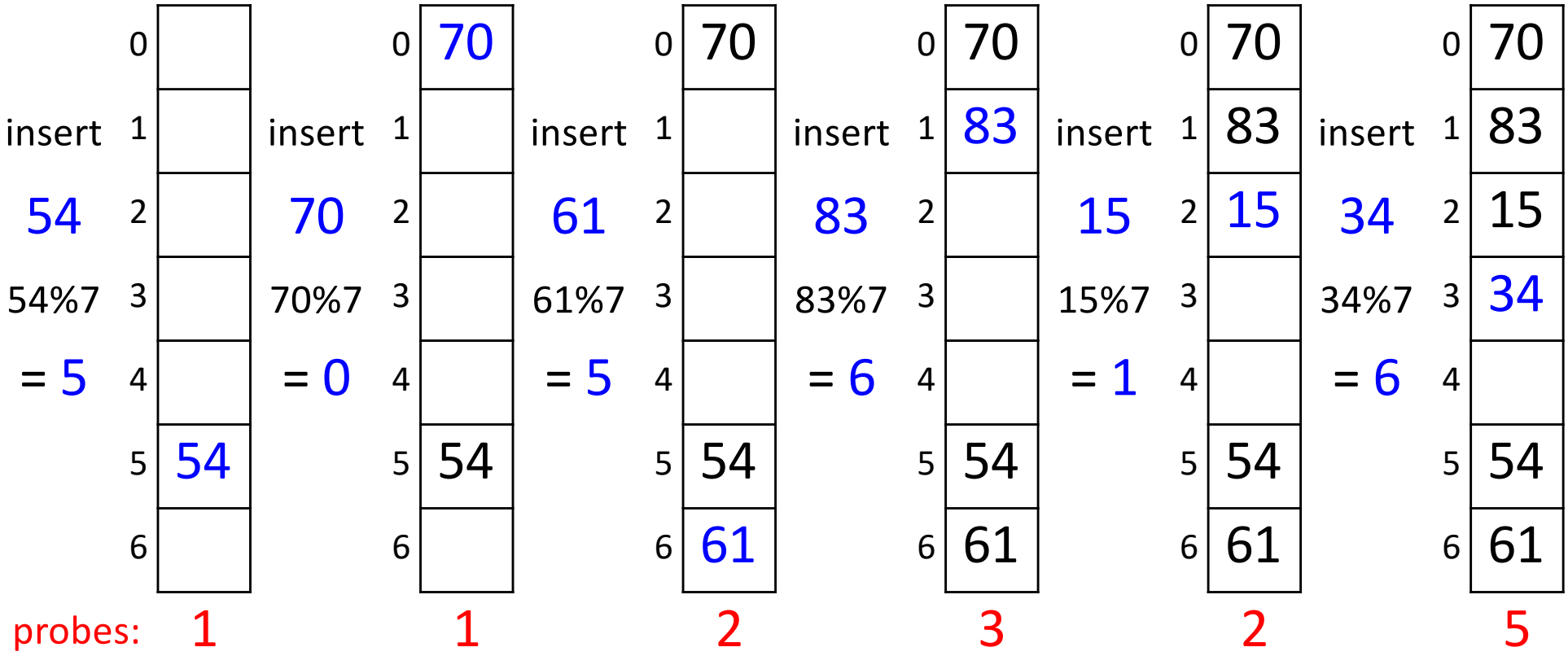


probes:

h(key) = key mod 7

Linear Probing

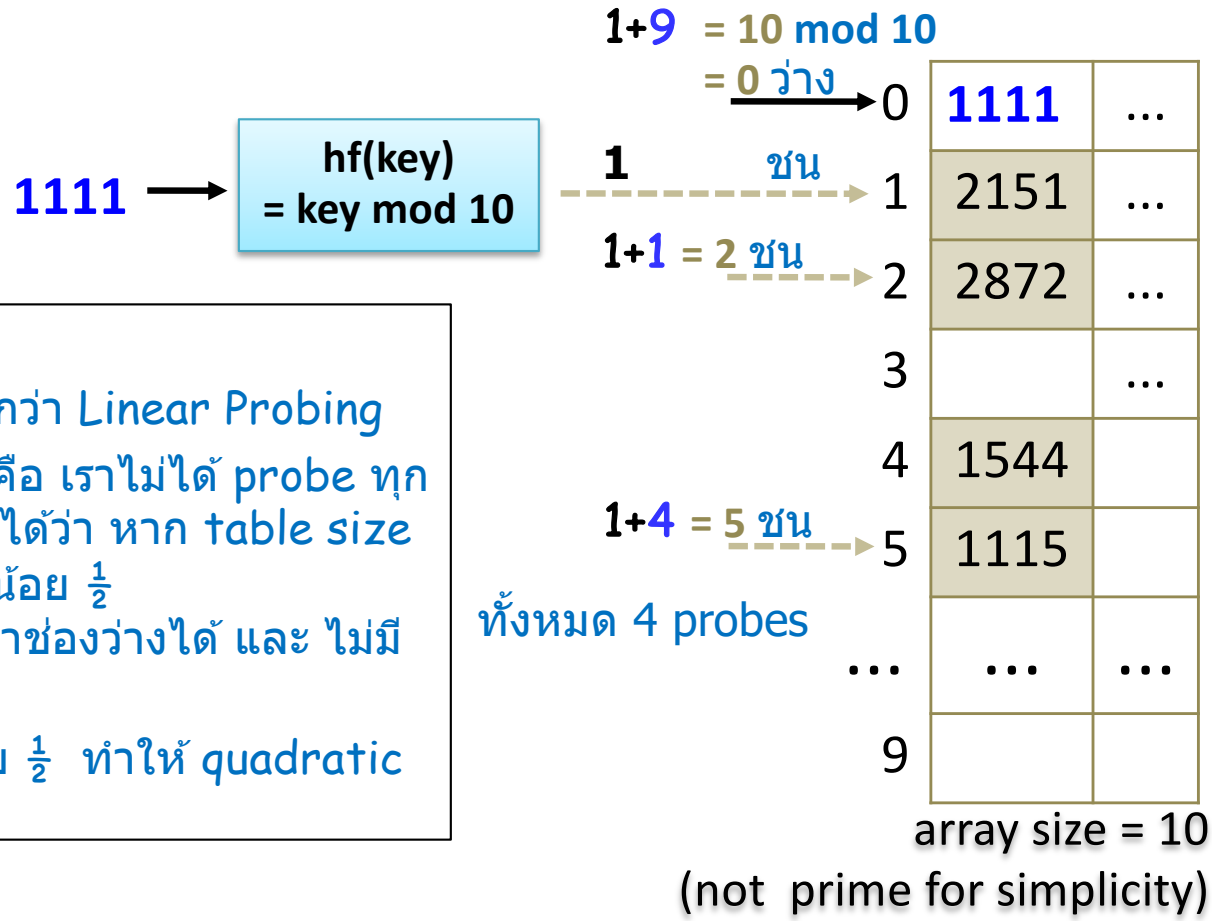
probe ชนครั้งที่ i	ลอง h(k) + f(i)
Linear Probing f(i) = i	ลอง : h(k), h(k)+1, h(k)+2, h(k)+3,...



h(key) = key mod 7

Quadratic Probing

probe ชนครั้งที่ 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,...

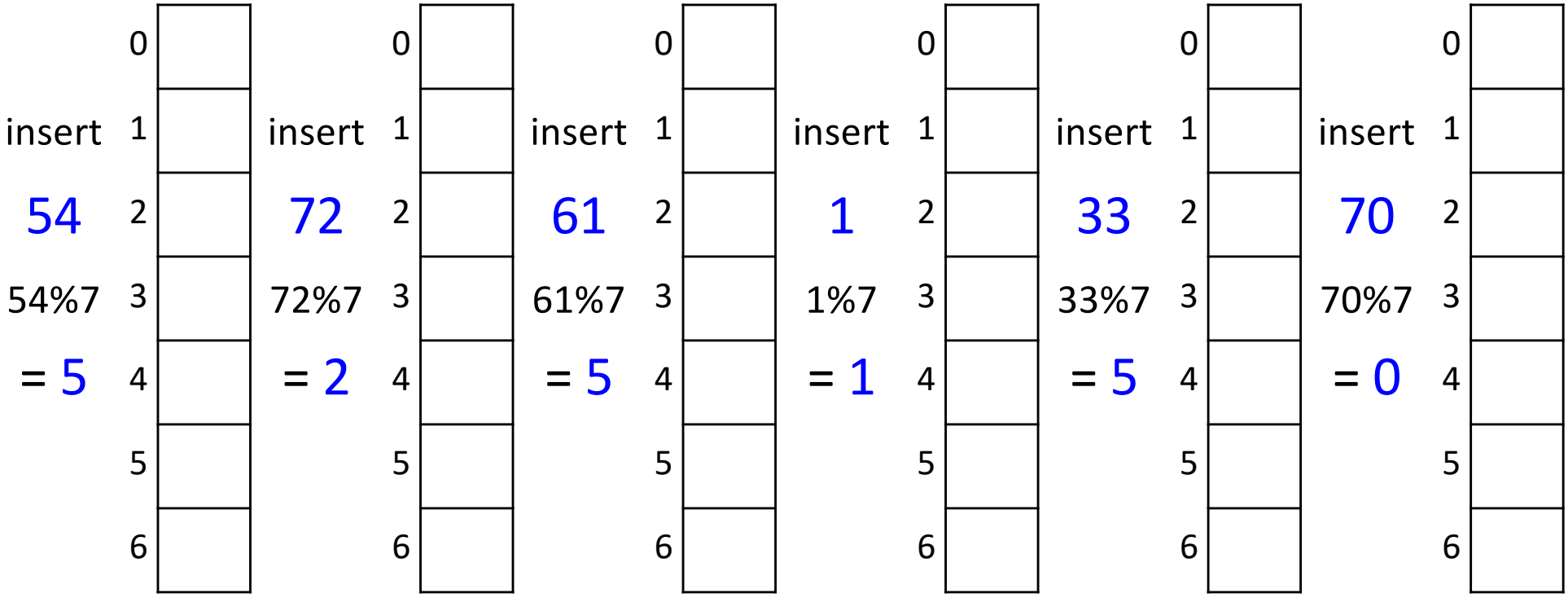


Quadratic Probing

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

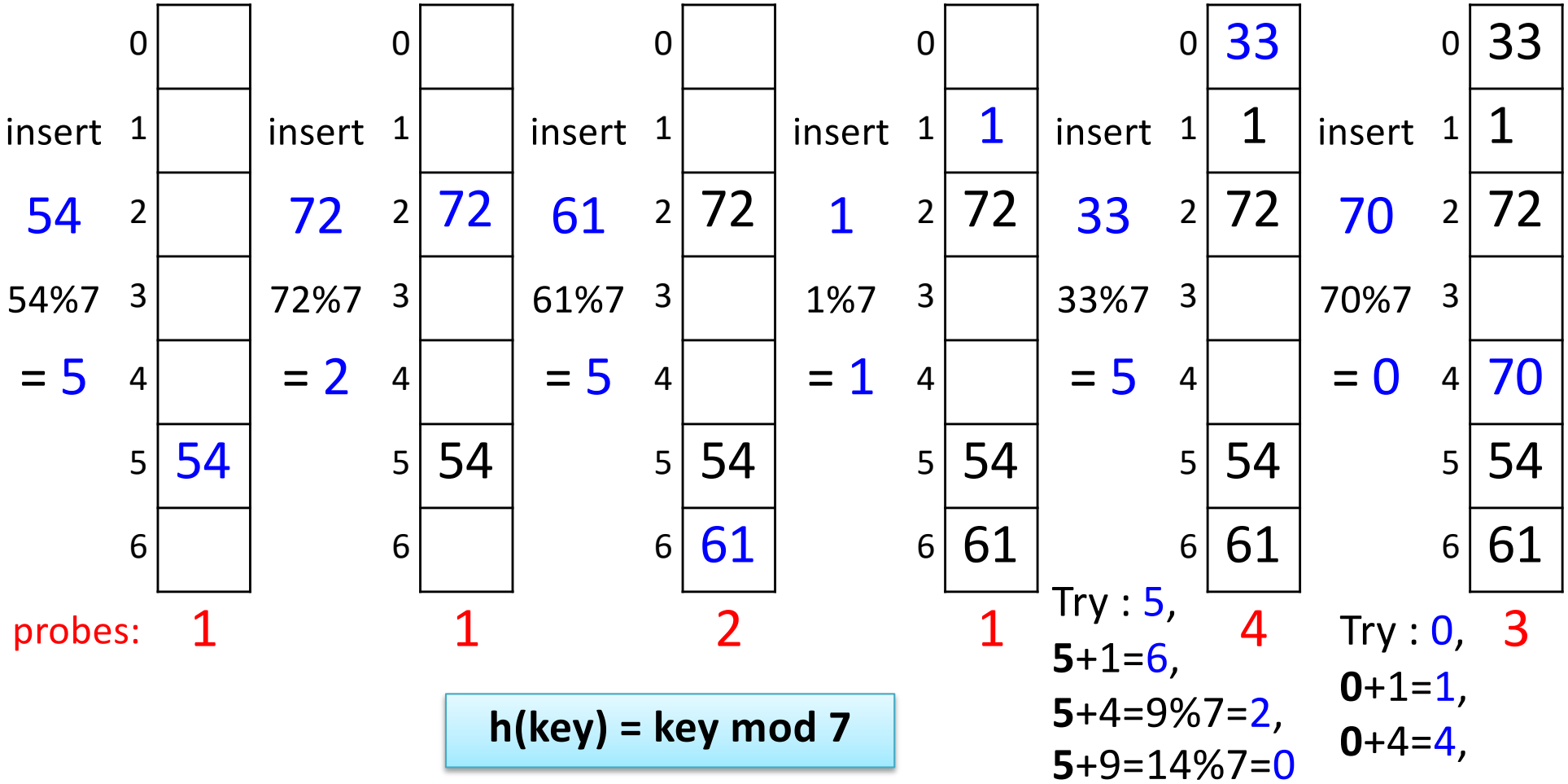


probes:

h(key) = key mod 7

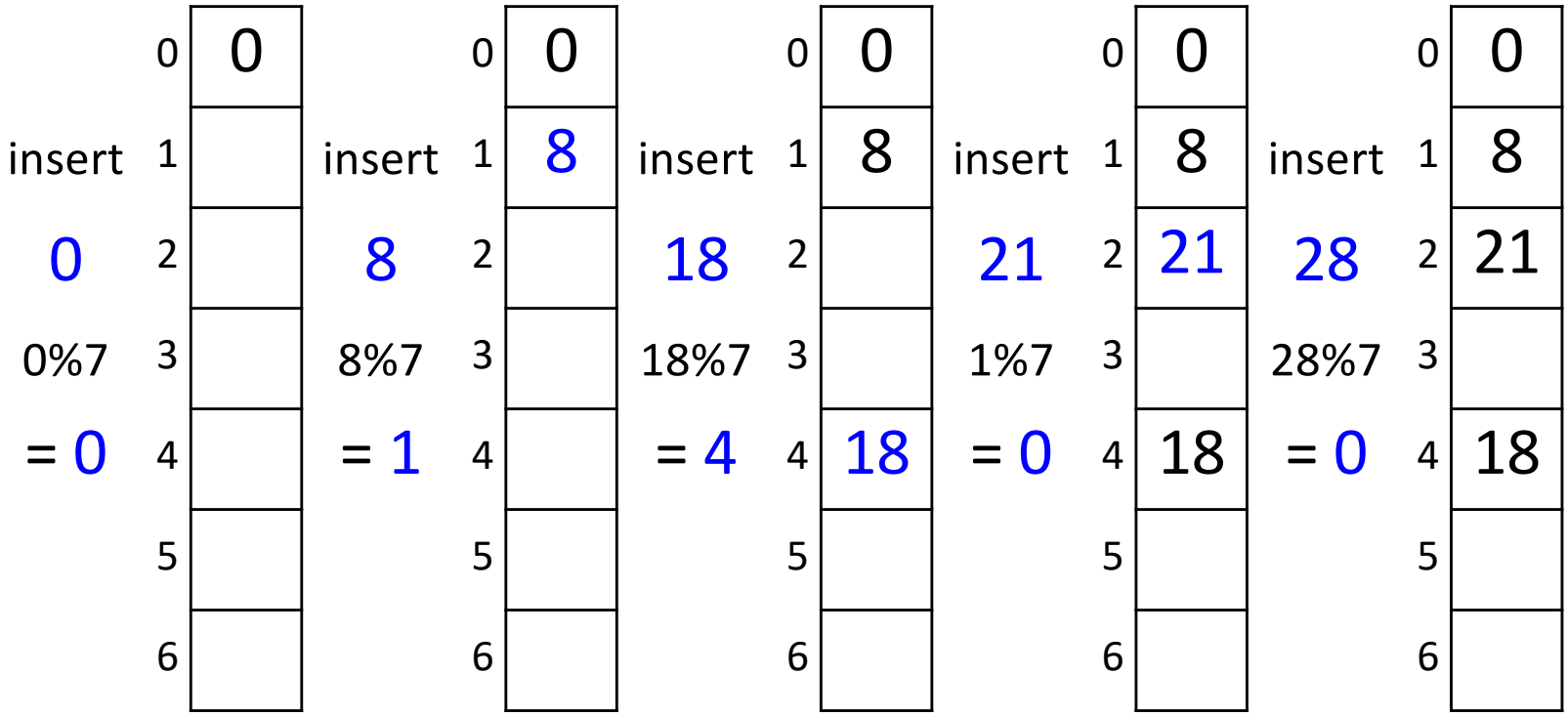
Quadratic Probing

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



Quadratic Probing

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



h(key) = key mod 7

Try : 0,
0+1=1,
0+4=4,
0+9=9%7=2

Try : 0,
0+1=1,
0+4=4,
0+9=9%7=2
0+16=16%7=2

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 เดิมทั้งหมดมา hash ใส่ table ใหม่ เรียกว่า rehashing (rehash ทุก data เข้า table ใหม่) (rehash อีกความหมายหนึ่งคือ hash อีกครั้งเมื่อ collission)
- เมื่อใดบ้างที่ต้อง rehash
  - ทันทีที่ table เต็ม
  - เมื่อ insert ไม่ได้ (นโยบายเข้มงวด)
  - table เต็มถึงระดับที่กำหนดไว้ (ถึงค่า load factor ที่กำหนด) (นโยบายเข้มงวดปานกลาง)

Rehashing

0	6
1	15
2	
3	24
4	
5	
6	13

Insert :
13, 15, 6, 24

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

Insert : 23
Over 70 % full

Rehashing

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

Rehashing →

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

h(x) = x mod 7, Linear Probing

ข้อเสียของการเก็บข้อมูลแบบ hash

- การจัดการที่เกี่ยวกับอันดับของข้อมูล
  - หาค่าตัวน้อยสุด
  - หาค่าตัวมากสุด
  - ตัวที่ถัดจากตัวที่กำหนดให้

  เนื่องจาก การต้องดูข้อมูลทุกตัวทุกตำแหน่งในตารางจึงจะตอบได้ และฟังก์ชันแฮชทำให้คีย์กระจาย ข้อมูลที่อยู่ใกล้กันเมื่อผ่านฟังก์ชันแฮชก็ไม่สามารถทราบได้ว่าอยู่ที่ใด
- ต้องระวังเรื่องฟังก์ชันแฮชที่ได้ค่าเหมือนกันทุกครั้ง
  - ในกรณีที่นำคีย์ผ่านฟังก์ชันแฮชแล้วได้ค่าซ้ำเดิม ซึ่งระบบจะแก้ปัญหาการชนไว้แล้ว แต่หากเพิ่มข้อมูลเข้าไปมาก ๆ ก็จะทำให้ระบบเริ่มช้าลงได้ จึงต้องระวังเรื่องการใช้งานฟังก์ชันแฮช

สรุป

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

สรุป

Complexity ของ Hash Table

Average case (กรณีทั่วไป เมื่อ hash function กระจายดี, load factor λ ไม่สูงเกินไป <0.7)

ค้นหา (Search) → O(1)
แทรก (Insert) → O(1)
ลบ (Delete) → O(1)

เพราะเข้าถึง index ใน array ได้โดยตรง

Worst case (กรณี hash function กระจายไม่ดี หรือ collision เยอะมาก ๆ เช่น key ทุกตัวชนกันหมด)

Search / Insert / Delete → O(n)

เพราะ key ทั้งหมดไปกองอยู่ที่ index เดียว ต้องไล่ list/array ทั้งหมด

Best case

ถ้าไม่มี collision เลย → O(1)