Lecturers :
Boontee Kruatrachue Room no. 913
Kritawan Siriboon Room no. 913

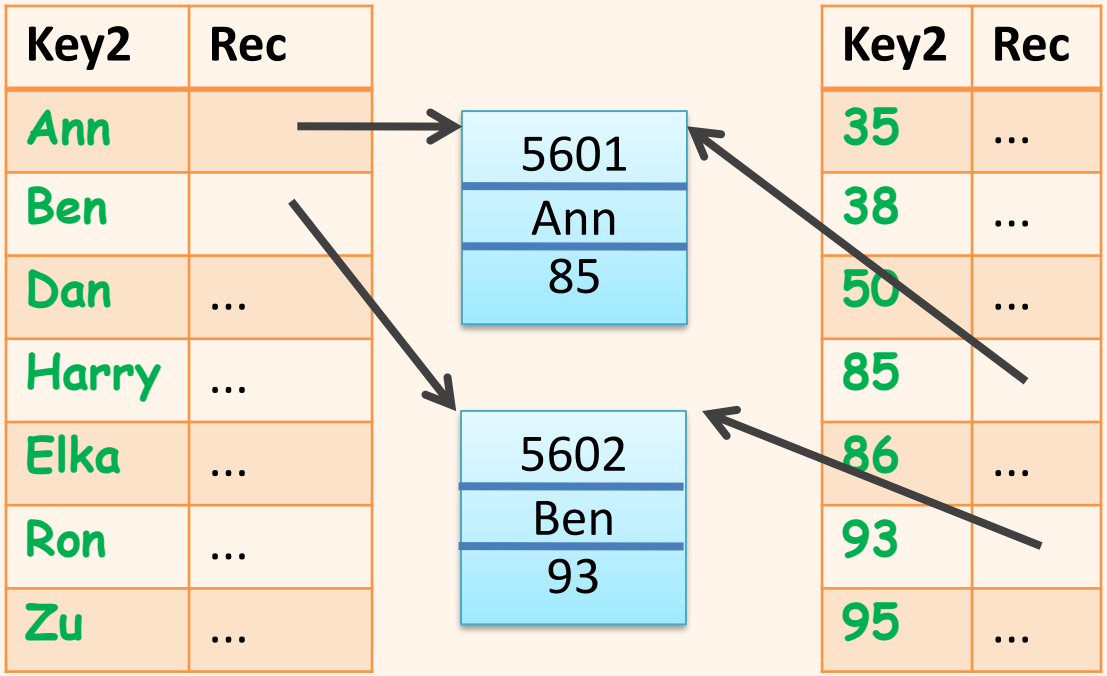
Searching

Searching Terminology

- Search, Record, Table/File
- Key
  - Internal Key (Embedded Key)
  - External Key

5601	5602	5603	...	...
Ann	Ben	Dan	...	...
85	93	50	...	...

record table/file



- Successful Search/Unsuccessful Search
- Internal Search / External Search

Searching

Search Unordered Table

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

Search Ordered Table

Array	List & Tree Search
- Sequential Search (Linear Search)	- Sequential Search (Linear Search)
- **Index Sequential Search**	- B-Tree
- Binary Search	-Binary Search Tree - AVL (Hight Balanced) Tree

- **Hashing**	**- Heap**

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
```

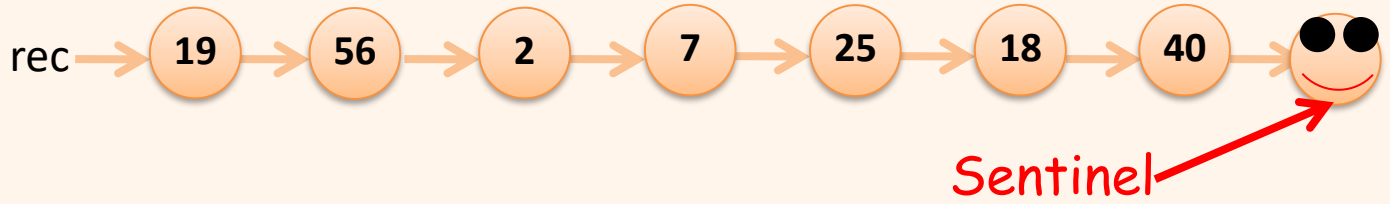
Sentinel Search

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

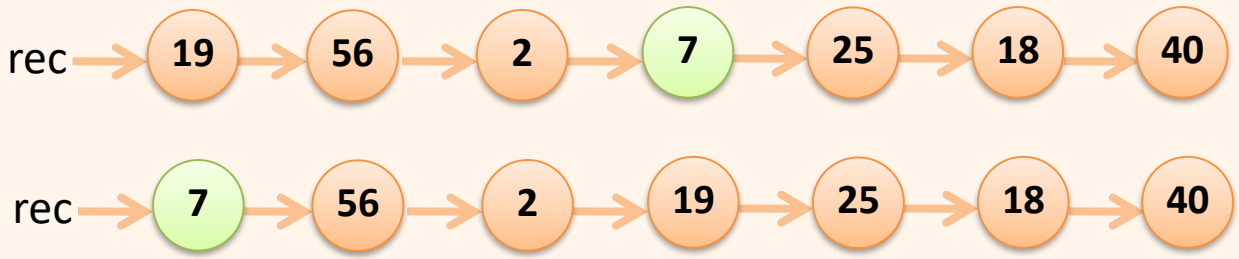
```
rec[n+1] = key;        //adding sentinel
i = 1;
loop (key <> rec[i].key)
    i = i+1;
end loop

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

O(n)

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



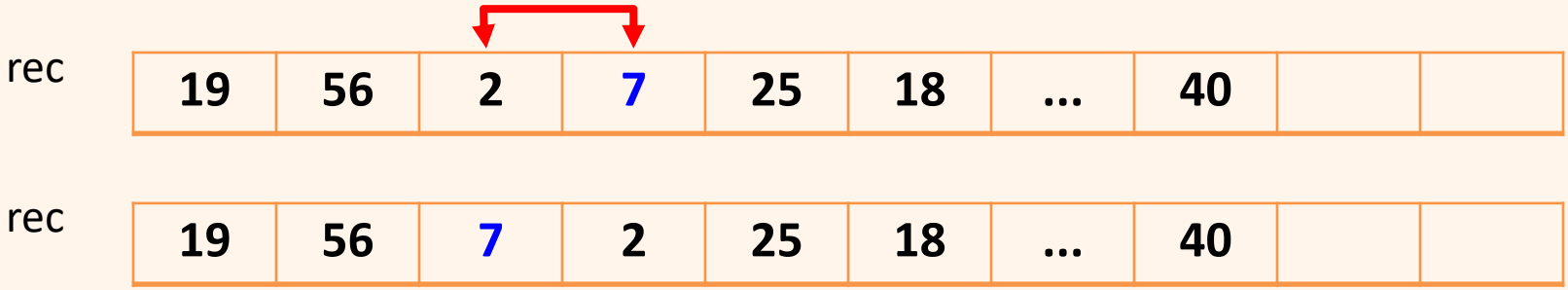
Move to Front Heuristic



Move to Front

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

Transposition Heuristic

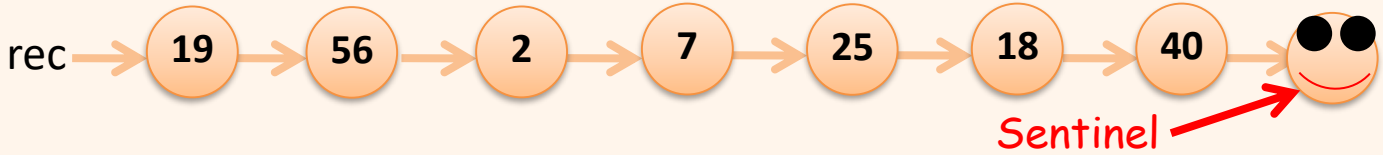


Transposition

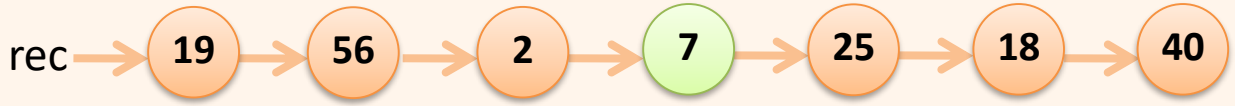
- แต่ละครั้งที่ search พบ ให้สลับ record ที่ search พบขึ้นมาข้างหน้า 1 ตำแหน่ง
- แนวคิด : การใช้ครั้งเดียวไม่ได้แปลว่าจะใช้อีกครั้งหนึ่งเสมอไป แต่การสลับแบบนี้หากใช้มากๆ ก็จะเลื่อนขึ้นมาอยู่ข้างหน้าเอง

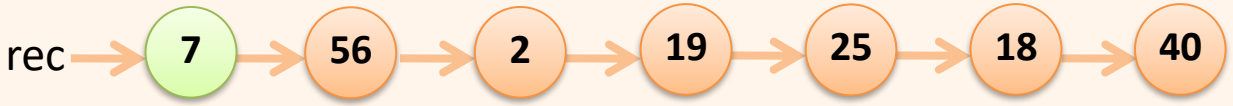
Searching Unordered Table

- **Sentinel Search**



- **Move to Front**





- **Transposition**

Searching

Search Unordered Table

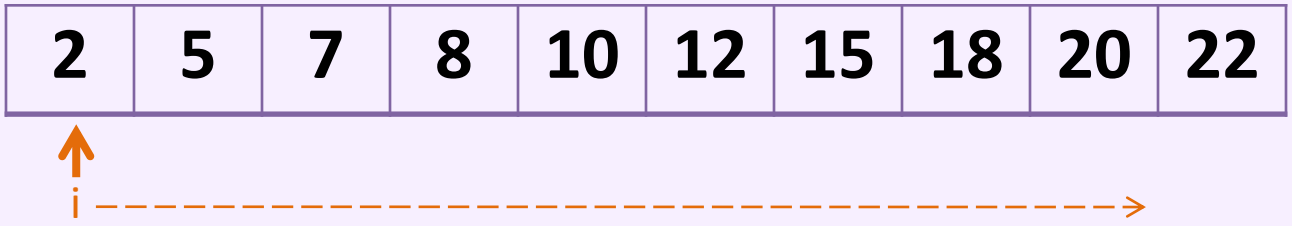
- Sentinel Search
- Move to Front
- Transposition

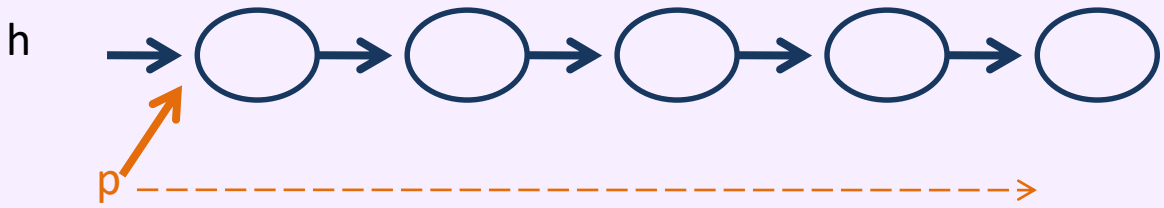
Search Ordered Table

Array	List & Tree Search
- Sequential Search (Linear Search)	- Sequential Search (Linear Search)
- Index Sequential Search	- B-Tree
- Binary Search	-Binary Search Tree - AVL (Hight Balanced) Tree

- Hashing	- Heap

Sequential Search (Linear Search)





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

Searching

Search Unordered Table

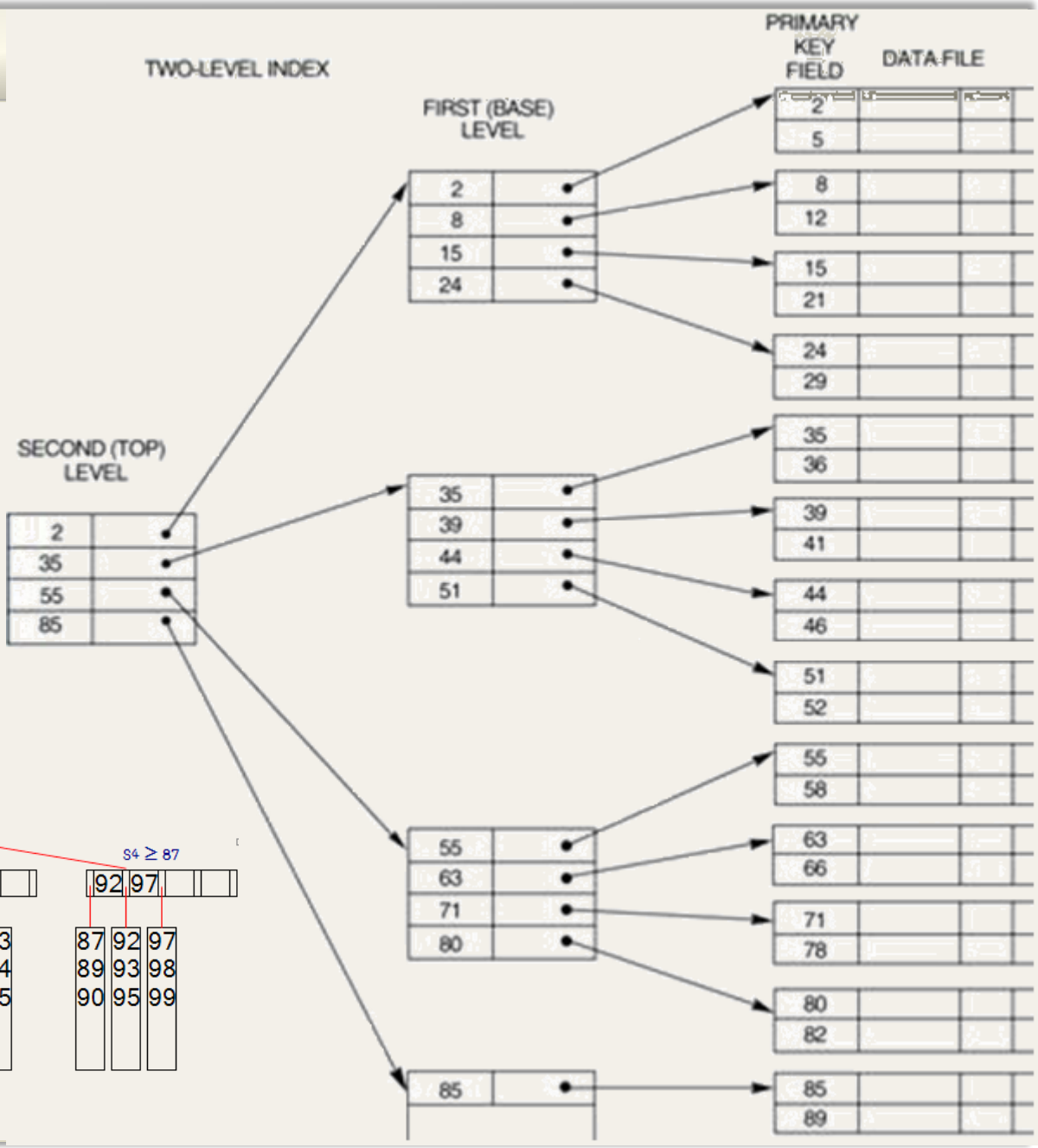
- Sentinel Search
- Move to Front
- Transposition

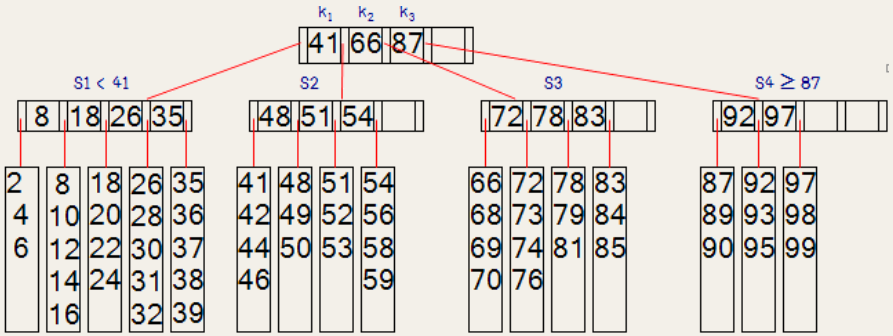
Search Ordered Table

Array	List & Tree Search
- Sequential Search (Linear Search)	- Sequential Search (Linear Search)
- **Index Sequential Search**	- B-Tree
- Binary Search	-Binary Search Tree - AVL (Hight Balanced) Tree

- **Hashing**	- **Heap**

Index Sequential Search





B-Tree Order 5

Searching

Search Unordered Table

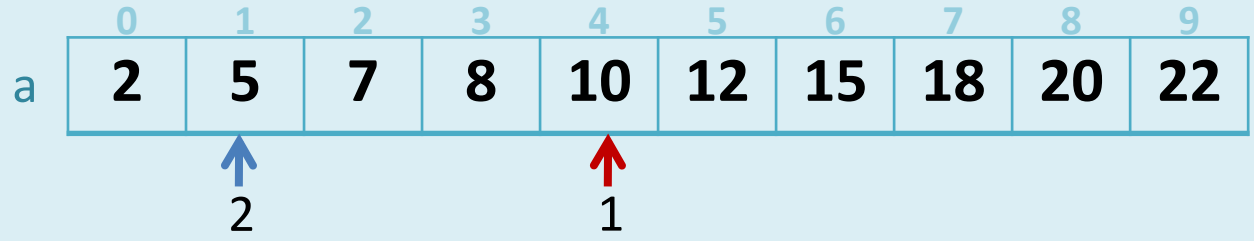
- Sentinel Search
- Move to Front
- Transposition

Search Ordered Table

Array	List & Tree Search
- Sequential Search (Linear Search)	- Sequential Search (Linear Search)
- Index Sequential Search	- B-Tree
- Binary Search	-Binary Search Tree - AVL (Hight Balanced) Tree

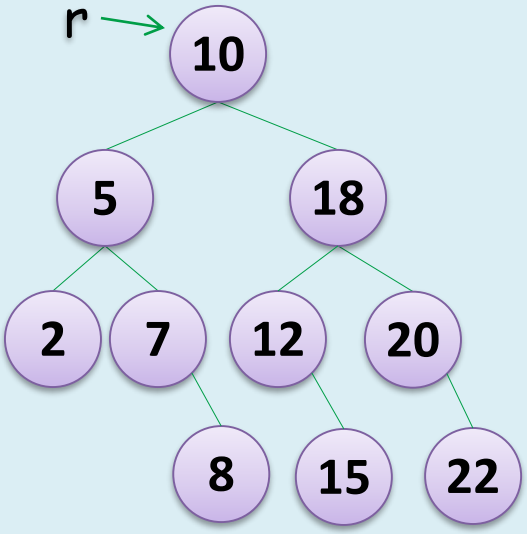
- Hashing	- Heap

Binary Search



binarySearch

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



$O(\log_2 n)$

binarySearch

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

Searching

Search Unordered Table

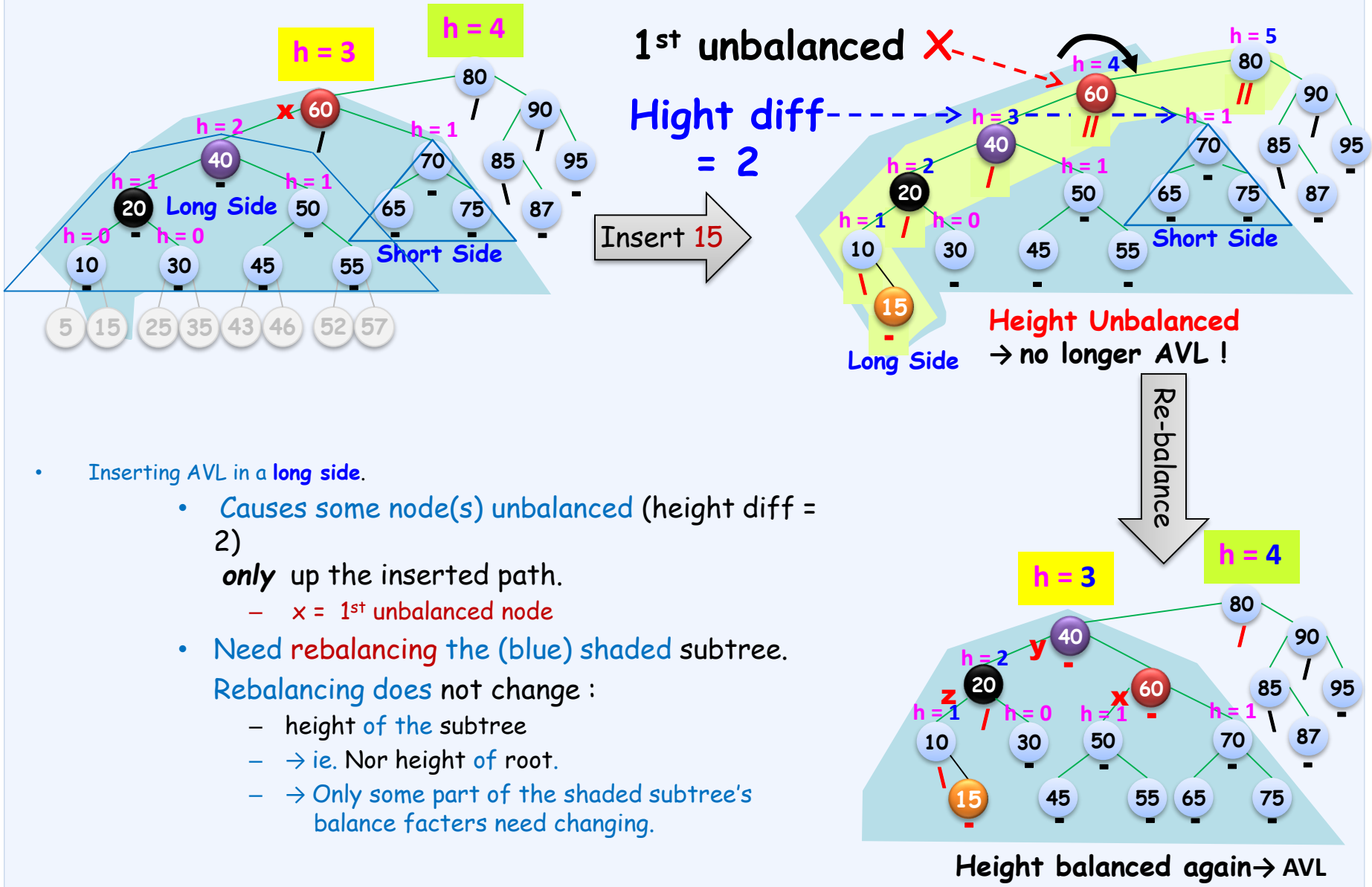
- Sentinel Search
- Move to Front
- Transposition

Search Ordered Table

Array	List & Tree Search
- Sequential Search (Linear Search)	- Sequential Search (Linear Search)
- Index Sequential Search	- B-Tree
- Binary Search	-Binary Search Tree - AVL (Hight Balanced) Tree

- Hashing	- Heap

Insertion Re-balancing in AVL (Height-Balanced) Tree



- Inserting AVL in a **long side**.
  - Causes some node(s) unbalanced (height diff = 2)
    only up the inserted path.
    - x = 1st unbalanced node
  - Need rebalancing the (blue) shaded subtree.
    Rebalancing does not change :
    - height of the subtree
    - → ie. Nor height of root.
    - → Only some part of the shaded subtree's balance facters need changing.

Searching

Search Unordered Table

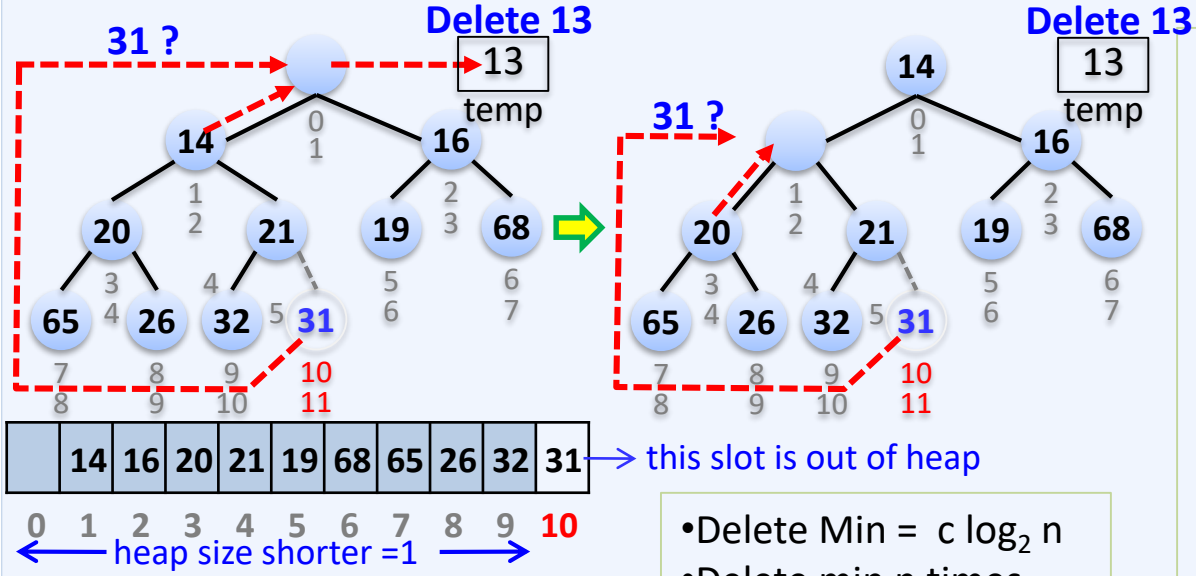
- Sentinel Search
- Move to Front
- Transposition

Search Ordered Table

Array	List & Tree Search
- Sequential Search (Linear Search)	- Sequential Search (Linear Search)
- Index Sequential Search	- B-Tree
- Binary Search	-Binary Search Tree - AVL (Hight Balanced) Tree

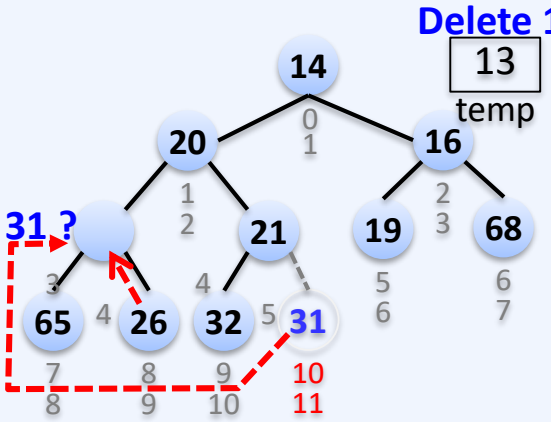
- Hashing	- Heap

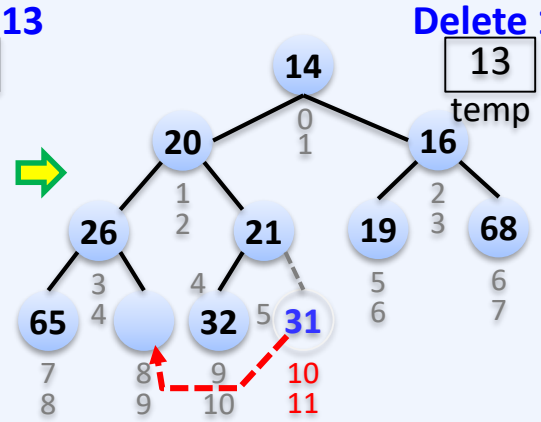
Heap : Delete Min

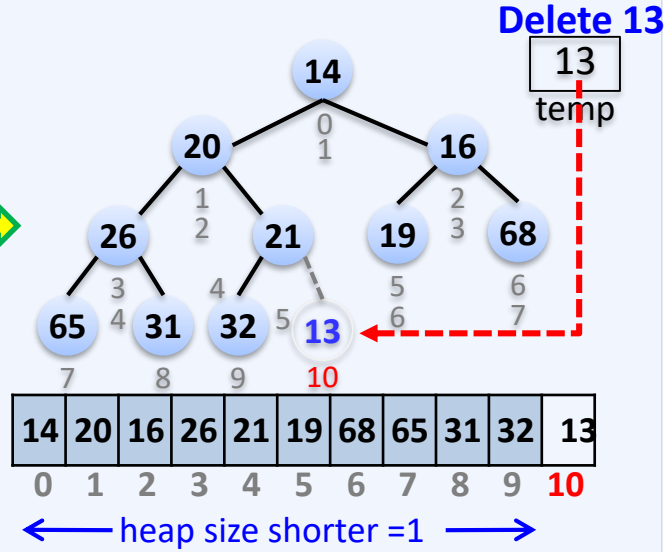


- Here is the min heap.
- Delete root gets min 13,hole at root.
- Heap size shorter 1: last node.
- Last node data,31, must be in the heap.
- Find place for 31 : reheap.
- Try the hole, if not, perlocate 31 down, take the hole's smaller son up.
- Repeat perlocate 31 down to find its place.
- Put the deleting data 13 at the last node place in the same array is called in place sort, makes decending order.
- Now 13 is out of heap.

- Delete Min = c $\log_2$ n
- Delete min n times.
- O(n $\log_2$ n)







Searching

Search Unordered Table

- Sentinel Search
- Move to Front
- Transposition

Search Ordered Table

Array	List & Tree Search
- Sequential Search (Linear Search)	- Sequential Search (Linear Search)
- Index Sequential Search	- B-Tree
- Binary Search	-Binary Search Tree - AVL (Hight Balanced) Tree

- **Hashing**	- Heap

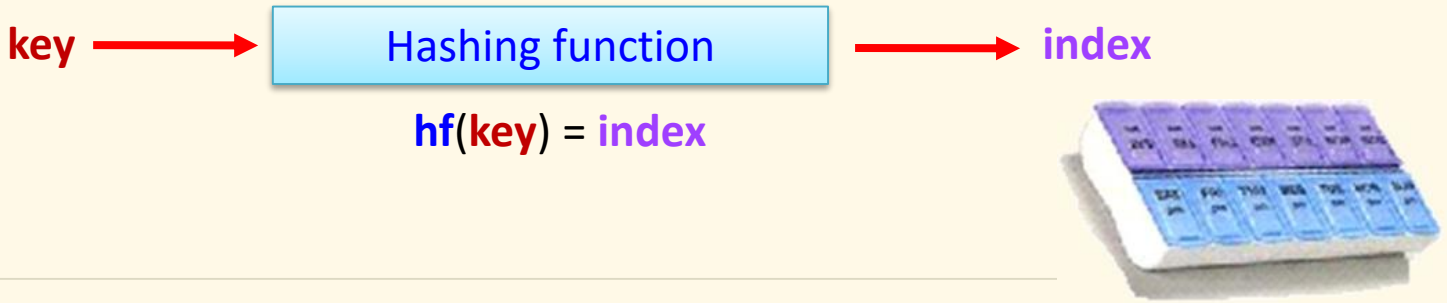
Hashing concepts

1. searches ก่อนๆ นี้ต้องเช็คเทียบหลายครั้งจึงพบหรือพบว่าไม่มี
   การเช็คเทียบแต่ละครั้ง เรียก probe
2. ความต้องการของ hashing :
   - ทำอย่างไรเราจะหาของได้ใน 1 probe ?
3. จะรู้ได้อย่างไรว่ามันอยู่ที่ไหน ?
4. มีของมาก เก็บอย่างไร ?

2. เราต้องรู้ว่ามันอยู่ที่ไหน
3. ต้องเก็บไว้เป็นที่
4. มีหลักการในการเก็บ

Hasing Function

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



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

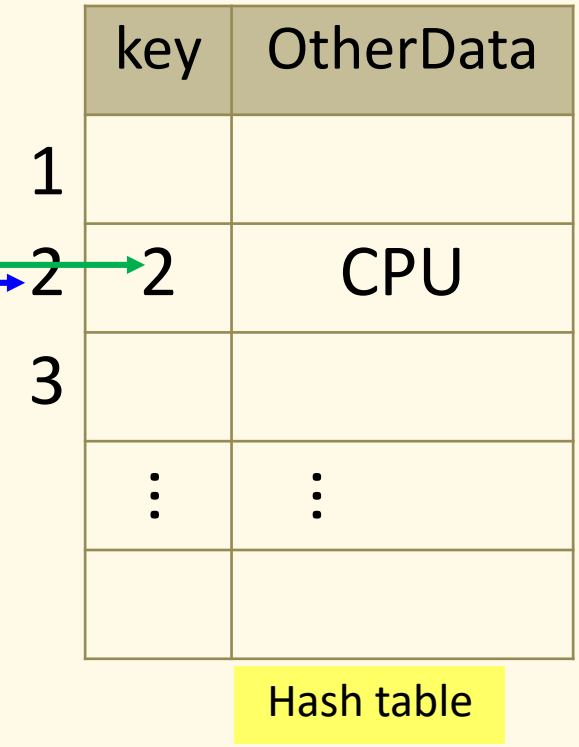
Inserting & Retriving Data

ถ้า key = index → สามารถ map ได้โดยตรง directly

hf(key) = key

- จะ insert record key = 2
  - hash 2 ด้วย hashing function ได้ 2
    hf(2) = 2
  - Insert record ใน slot index = 2

- จะ retrieve record key = 2
  - hash 2 with ด้วย hashing function ได้ 2
    hf(2) = 2
  - ไปหา record ใน slot index = 2



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 -> 4156

Summation, Division, ...

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

    54091576
    = 540 +915+76
    = 1531
    = 531 //array size = 1000

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

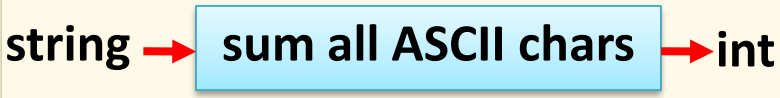
    key^2 หรือ $part\_of\_key^2$

    เอาตรงกลางมา

    54011576 -> 576*576
    = 331776 -> 177

Mapping String 1 - 2

Mapping String 1

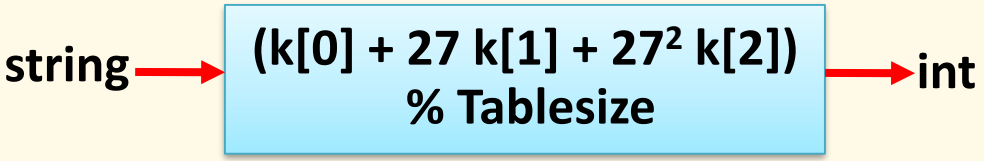


HashString1

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

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

```
HashString3
   k = address of first char
   loop (*k != null)
       hv = (hv<<5) + *k++
   end loop
   hv = hv % Tablesize
   return  hv
```

// hv<<1 = hv*2,
// hv<<5=hv*2^5=hv*32

Mapping String 3
Horner's rule : Polynomial of 32

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

Mapping String

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

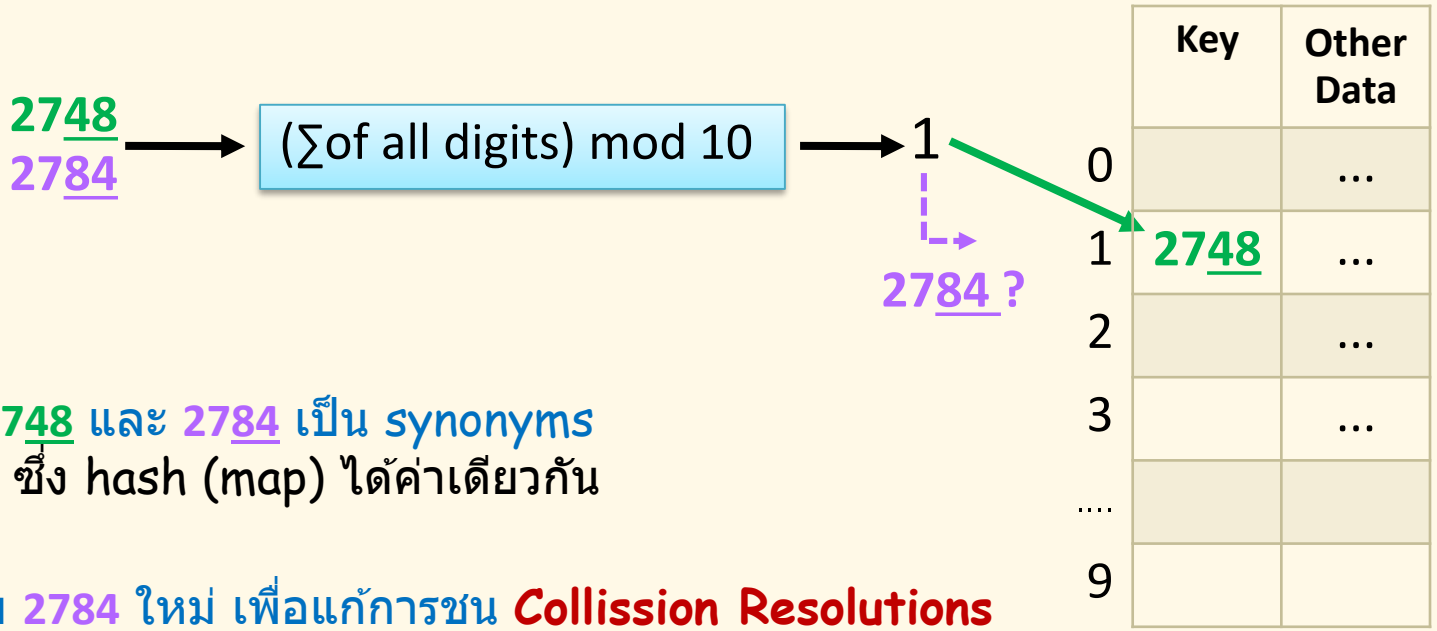
  = 438WaNY

  map -> array range

Collision, Synonym

Collision 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



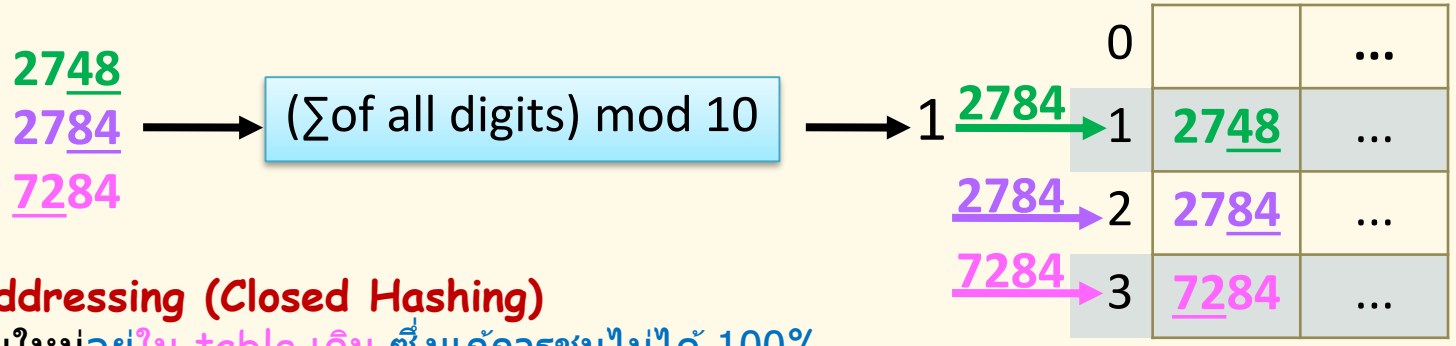
Synonyms : 2748 และ 2784 เป็น synonyms
set of keys ซึ่ง hash (map) ได้ค่าเดียวกัน

- ต้องหาที่เก็บ 2784 ใหม่ เพื่อแก้การชน **Collission Resolutions**
  - **Open Addressing**
  - **Seperate Chainning**

Hashing Function ที่ดี

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

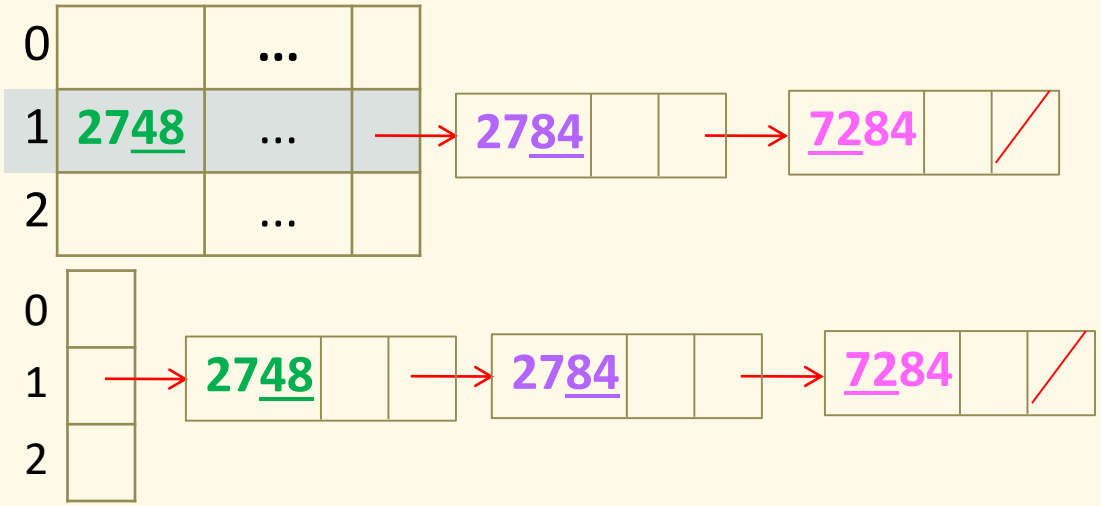
Collission Resolutions



Open Addressing (Closed Hashing)

ที่เก็บใหม่อยู่ใน table เดิม ซึ่งแก้การชนไม่ได้ 100%
เช่น2784 ชนกับ 2748 จึงหาที่ใหม่ให้ ในตัวอย่าง
- ให้อยู่ถัดมา 1 ช่อง วิธีนี้เรียกว่า **Linear Probing**

Seperate Chaining ที่เก็บใหม่อยู่นอก table เดิม แก้การชนได้ 100%



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

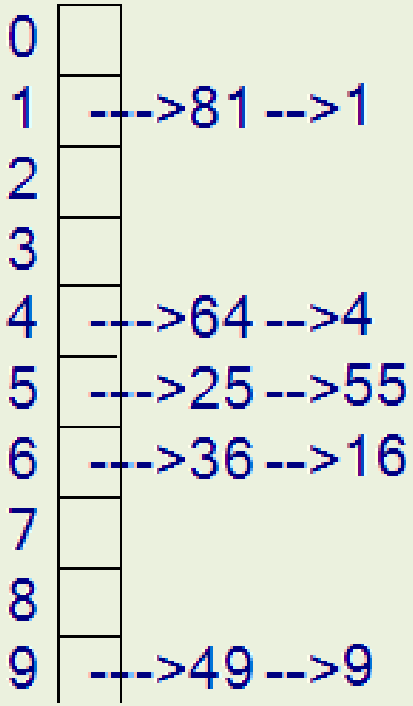
Seperate Chaining

ที่เก็บใหม่อยู่นอก table เติม แก้การชนได้ 100%

Seperate Chaining (cont.)

- **Load factor λ**
  = จำนวน element ใน hash table / table size
  - ในรูปตัวอย่าง λ = 10/10 = 1.0
- จากการวิเคราะห์ : table size ไม่ค่อยสำคัญเท่าไหร่ ที่สำคัญคือ λ
- general rule สำหรับ separate chaining hash table
  - λ ~ 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 collission

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



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,...

h(1111) = 1 ชน

probe sequence :

1, 1+1, 1+2, 1+3,...

1 2 3 ชน

4 ว่าง

ทั้งหมด 4 probs

Rehash: hash อีกครั้งเพื่อให้ได้ที่ใหม่

1111 → **hf(key) = key mod 10**

1 ชน

1+1 = 2 ชน

1+2 = 3 ชน

1+3 = 4 ว่าง

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

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,...

h(1111) = 1 ชน

probe sequence :

1, 1+1, 1+4, 1+9,...

1 2 5 ชน

10 mod 10 = 1 ว่าง

ทั้งหมด 4 probs

1111 → **hf(key) = key mod 10**

1+9 = 10 mod 10 = 1 ว่าง → 0

1 ชน → 1

1+1 = 2 ชน → 2

1+4 = 5 ชน → 5

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

Open Addressing (Closed Hashing)

Open Addressing : พยายาม probing จนพบที่ว่าง สำหรับ key k

- 1st probe — ลอง h(k)
- 2nd probe (ชนครั้งที่ 1) — ลอง h(k) + f(1)
- 3rd probe (ชนครั้งที่ 2) — ลอง h(k) + f(2)
- ith probe (ชนครั้งที่ i) — ลอง h(k) + f(i)

ith 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,...
• Double Hashing	2 hash functions	

Primary & Secondary Clusterings

Collission Resolutions

- **Open Addressing (Closed Hashing)** ที่เก็บใหม่อยู่ใน table เดิม ซึ่งแก้การชนไม่ได้ 100%
  - **Linear Probing, Quadratic Probing, ...** แก้ได้เฉพาะ **Primary Clustering**
  - **Double Hashing** แก้ **Secondary Clustering** ได้
- **Seperate Chainning** ที่เก็บใหม่อยู่นอก table เดิม แก้การชนได้ 100%

Primary Clustering

ปรากฏการณ์ที่ ครั้งแรกที่ชน hash คนละ key ไปตกที่คนละ slot
แต่ต้องมาแย่งที่กันในที่สุดในการ rehash ครั้งถัดๆมา

Secondary Clustering

ปรากฏการณ์ที่ ครั้งแรกที่ชน hash คนละ key ไปตกที่ slot x เดียวกัน

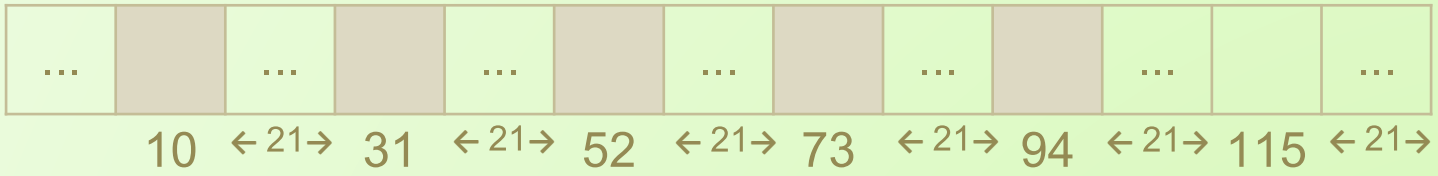
- hash function ฟังก์ชั่นเดียว ซึ่งใช้ค่า x ไปคำนวณที่เก็บใหม่ ได้ค่า series เดียวกันไปตลอด
- ต้องใช้ hash function ที่ 2 มาช่วยให้ไปตกที่ที่ใหม่ที่แตกต่างจาก hash function ที่ 1
- เรียกวิธีนี้ว่า **Double Hashing**

Primary Clustering

Primary Clustering ปรากฏการณ์ที่ ครั้งแรกที่ชน hash คนละ key ไป**ตกที่คนละ slot** แต่ต้องมาแย่งที่กันในที่สุดในการ rehash ครั้งถัดๆมา

Solutions: ให้ rehash f^n ขึ้นกับจำนวนครั้งที่ rehash ด้วย จะได้คนละ path เช่น

Linear probing rhi = (h(k)+i) mod ... // rehash ครั้งที่ i



Primary Clustering: แย่ง slot 115

h(k) = (k+21) mod ... rh(i) = (i+21) mod ...

//i = last hash value

Prob Sequences:

key 10 : 31 52 73 94 115
key 31 : 52 73 94 115
key 52 : 73 94 115
key 73 : 94 115
key 94 : 115

Linear Probing: ได้คนละ series ไม่แย่ง 115 กัน

Prob Sequences:

key 10 : 31 32 33 34 35
key 31 : 52 53 54 55
key 52 : 73 74 75
key 73 : 94 95
key 94 : 115

Primary Clustering Solutions

Primary Clustering ปรากฏการณ์ที่ ครั้งแรกที่ชน hash คนละ key ไปตกที่คนละ slot แต่ต้องมาแย่งที่กันในที่สุดในการ rehash ครั้งถัดๆมา

Solutions: ให้ rehash f^n ขึ้นกับจำนวนครั้งที่ rehash ด้วย จะได้คนละ path

1. **Linear probing** $rh_j = (h(k)+j) \bmod \ldots$ // rehash ครั้งที่ j

   key 10 : 31 32 33 34

   key 31 : 52 53 54 55

2. rh(i, j) = (i + j) mod ... //rehash i ครั้งที่ j

   $rh_1 = (h(k) + 1) \bmod \ldots$ //rehash ครั้งที่ 1

   $rh_2 = (rh_1 + 2) \bmod \ldots$ //rehash ครั้งที่ 2

   $rh_3 = (rh_2 + 3) \bmod \ldots$ //rehash ครั้งที่ 3

   key 10 : 31 32 34 37

   key 31 : 52 53 55 58

* ทุกวิธีให้ **path** ที่ต่างกัน
* ทุกวิธีไม่สามารถกำจัด **secondary clustering** ได้

Primary Clustering Solutions

เมื่อ rehash ครั้งที่ j

3. **Quadratic probing** rh_j = (h(k)+sqr(j))) mod ...

   key 10 : 31 32 35 40

   key 31 : 52 53 56 61

4. rh(i,k) = (i + hkey) mod ... // เมื่อ hkey = 1+h(k) mod...

5. ใช้ random permutation ของเลขใดๆในช่วง 1..max p1, p2, p3, ...

   rh_j = h(k) + p_j) mod ...

Secondary Clustering
Double Hashing

Secondary Clustering

ปรากฏการณ์ที่ ครั้งแรกที่ชน hash คนละ key ไปตกที่ slot x เดียวกัน

- hash function ฟังก์ชั่นเดียว ใช้ค่า x ไปคำนวณที่เก็บใหม่ ได้ค่า series เดียวกันไปตลอด
- ต้องใช้ hash function ที่ 2 มาช่วยให้ไปตกที่ที่ใหม่ที่แตกต่างจาก hash function ที่ 1
- เรียกวิธีนี้ว่า **Double Hashing**

Double Hashing : 2 hash functions เช่น

1. ใช้ primary function i = h1(k) ถ้าไม่ว่าง
2. rh(j, k) = (h1(k) + j * h2(k)) mod ... ไปเรื่อยๆ //เมื่อ rehash ครั้งที่ j

ตราบใดที่ h2(k) ≠ h1(k) ไม่ collision

อย่างไรก็ดี ไม่สามารถแก้ collision ได้ 100% แก้ 100% -> chaining

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 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)

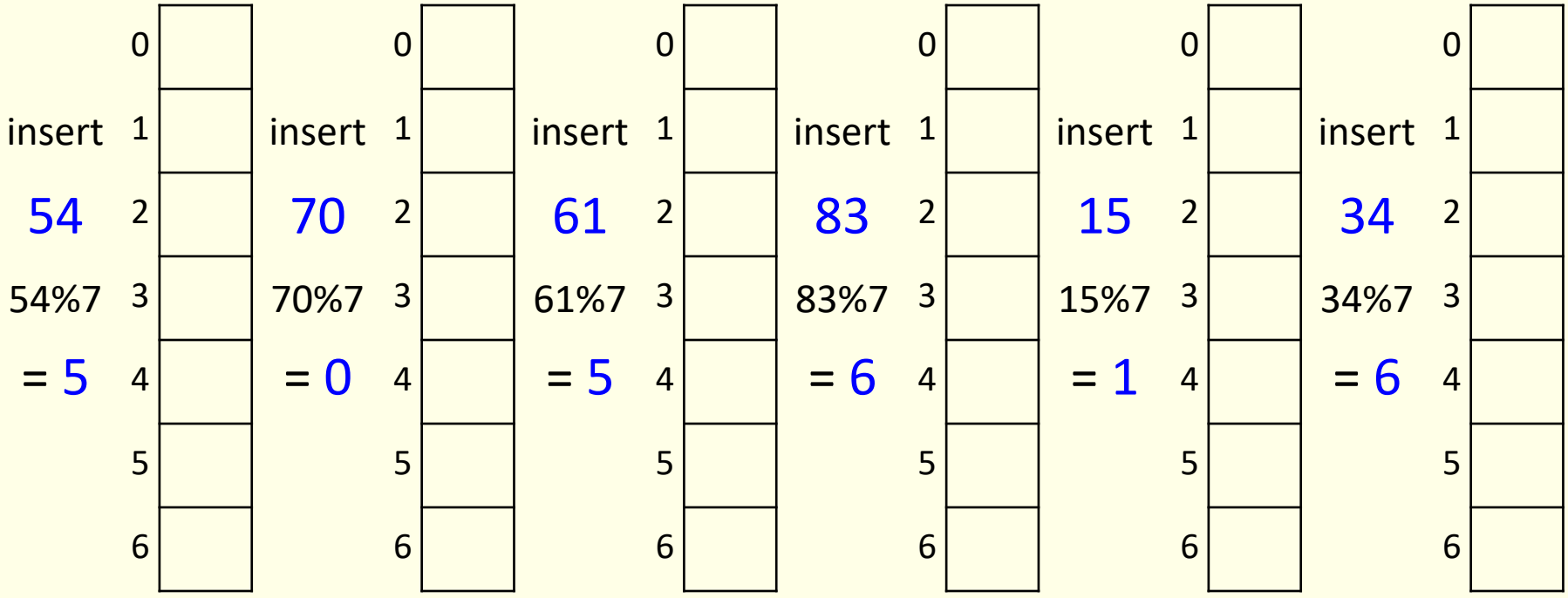
Rehashing →

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

Linear Probing

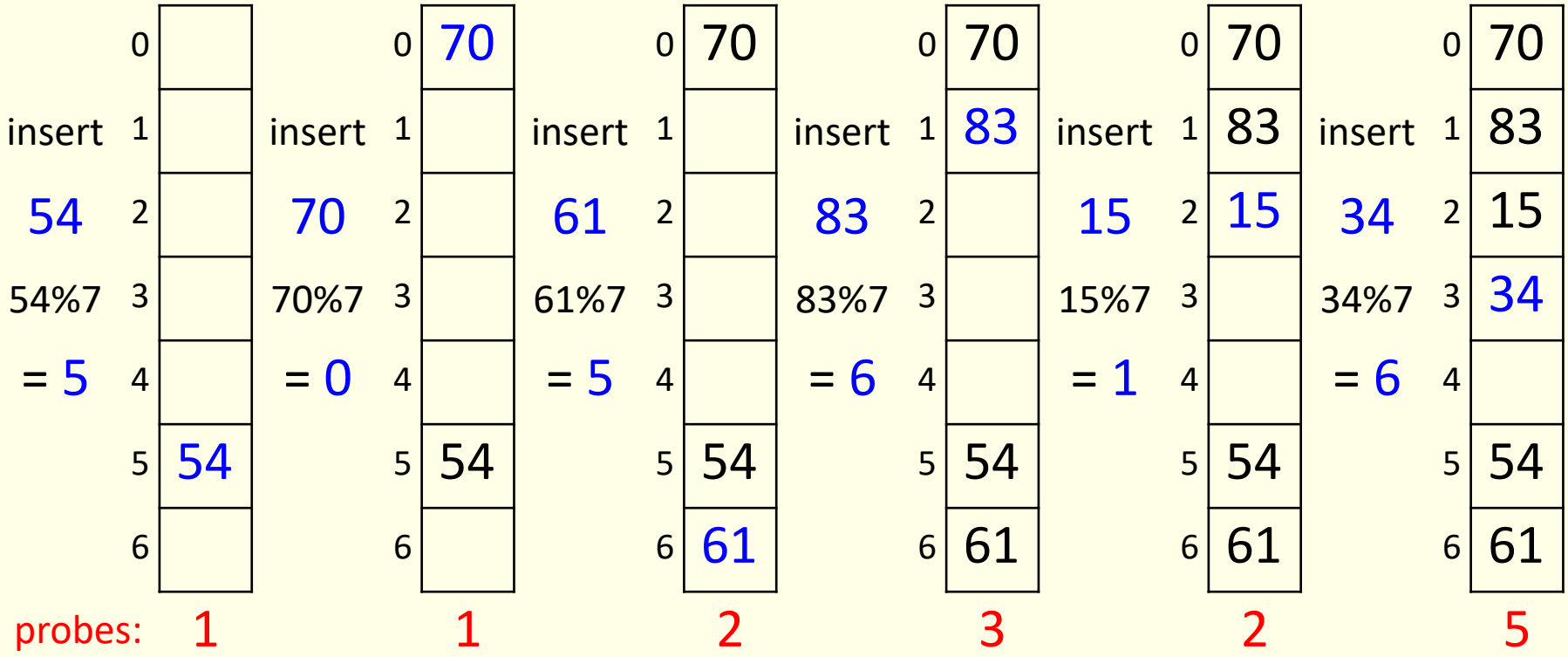
probe ชนครั้งที่ i	ลอง	h(k) + f(i)
Linear Probing f(i) = i	ลอง :	h(k), h(k)+1, h(k)+2, h(k)+3,...



h(key) = key mod 7

Linear Probing

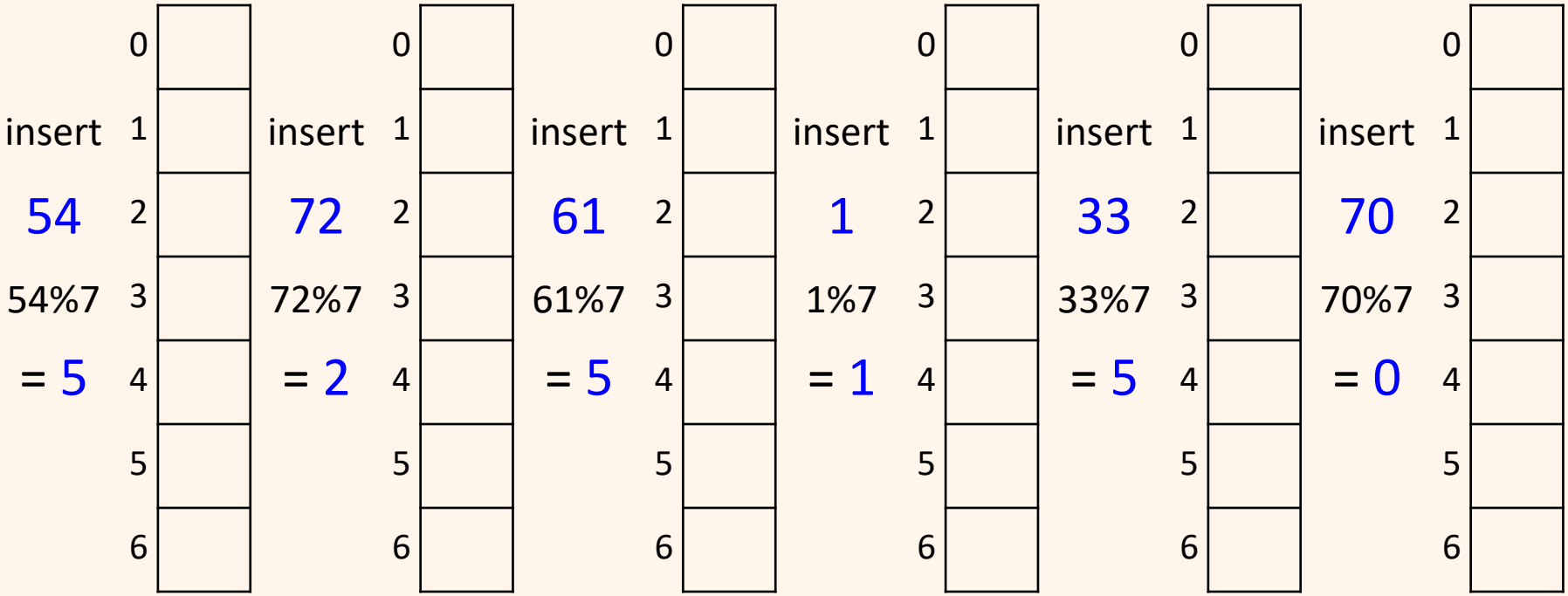
probe ชนครั้งที่ i		ลอง	h(k) + f(i)
Linear Probing	f(i) = i	ลอง :	h(k), h(k)+1, h(k)+2, h(k)+3,...



h(key) = key mod 7

Quadratic Probing

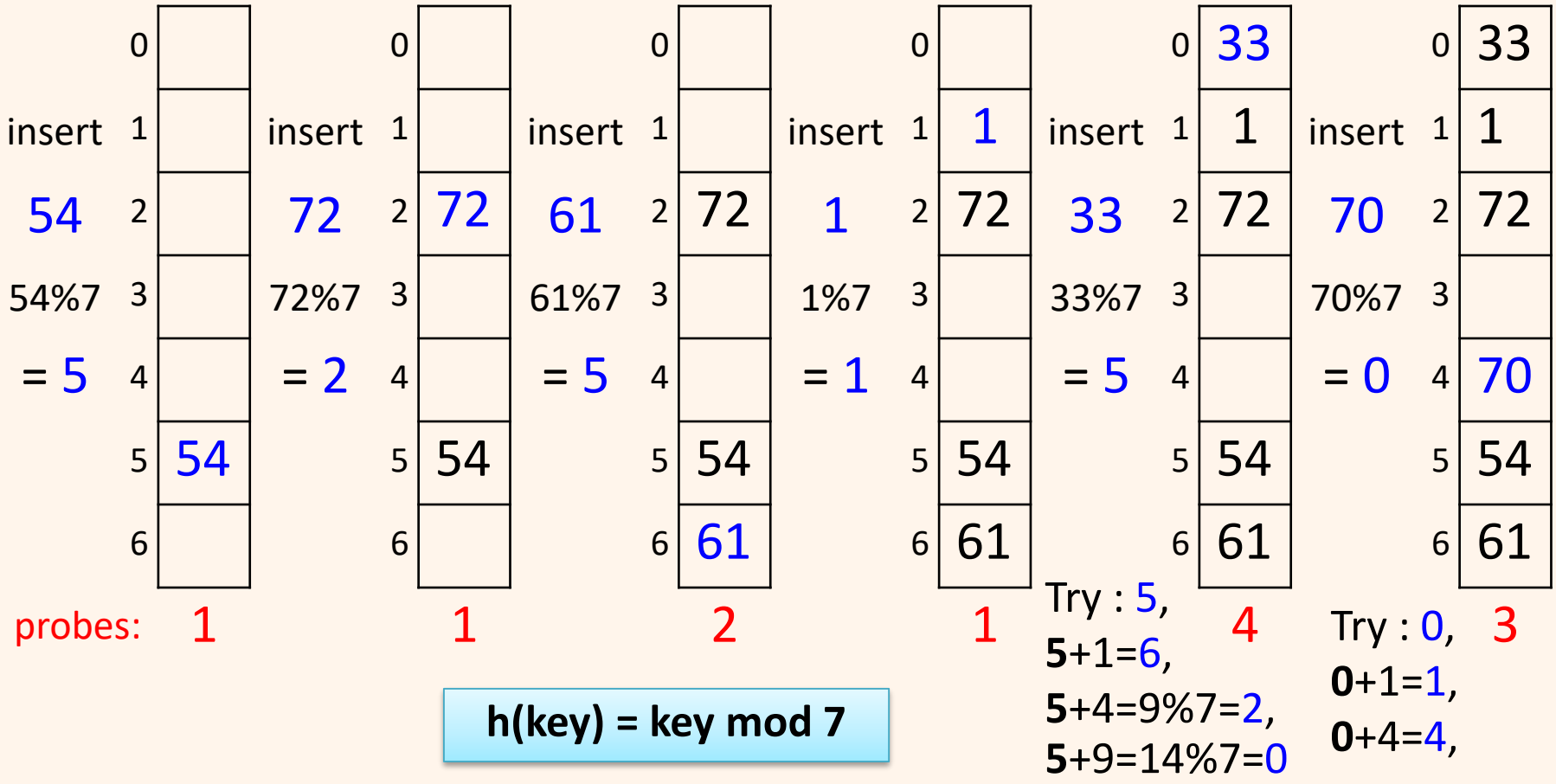
probe ชนครั้งที่ i	ลอง	h(k) + f(i)
Quadratic Probing f(i) = i^2	ลอง :	h(k), h(k)+1, h(k)+4, h(k)+9,...



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,...



Done