Chapter 8

Searching

Searching

- เราสามารถใช้คอมพิวเตอร์มาช่วยประมวลผลงานทั่วๆไป เช่น การเก็บข้อมูลต่างๆ ซึ่งมักมีขนาดใหญ่ และมีความจำเป็นต้องเรียกใช้เสมอ
- ผู้ใช้ต้องสามารถจัดการข้อมูล ทั้งการเพิ่ม, แก้ไข หรือลบข้อมูลที่มีได้อย่างรวดเร็ว
- นั่นคือ จำเป็นต้องมีอัลกอริทึมที่ช่วยในการค้นหา (Searching) ข้อมูลได้อย่างรวดเร็ว ซึ่งจะทำให้การจัดการข้อมูลต่างๆ มีประสิทธิภาพมากยิ่งขึ้น

Basic Search Algorithm

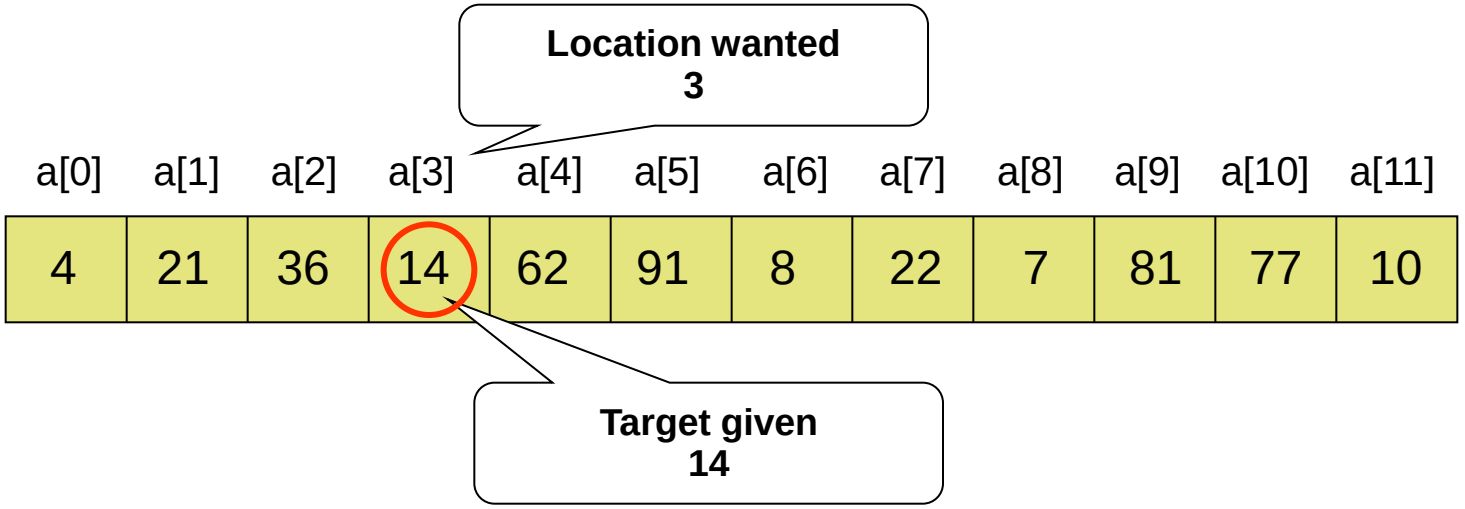
- Sequential Search
- Binary Search
- Hashing

Sequential Search

- เป็นการค้นหาข้อมูลในลักษณะเรียงลำดับ โดยไล่เปรียบเทียบตั้งแต่ข้อมูลตัวแรก ตัวที่สอง ไล่ไปเรื่อยๆ จนกว่าจะพบข้อมูลที่ต้องการ
- ในกรณีที่ ไล่เปรียบเทียบจนถึงข้อมูลตัวสุดท้ายแล้วยังไม่พบข้อมูลที่ต้องการ แสดงว่าไม่มีข้อมูลดังกล่าว

Sequential Search (Cont.)

- ต้องการค้นหาข้อมูลที่มีค่า 14 จากชุดข้อมูล

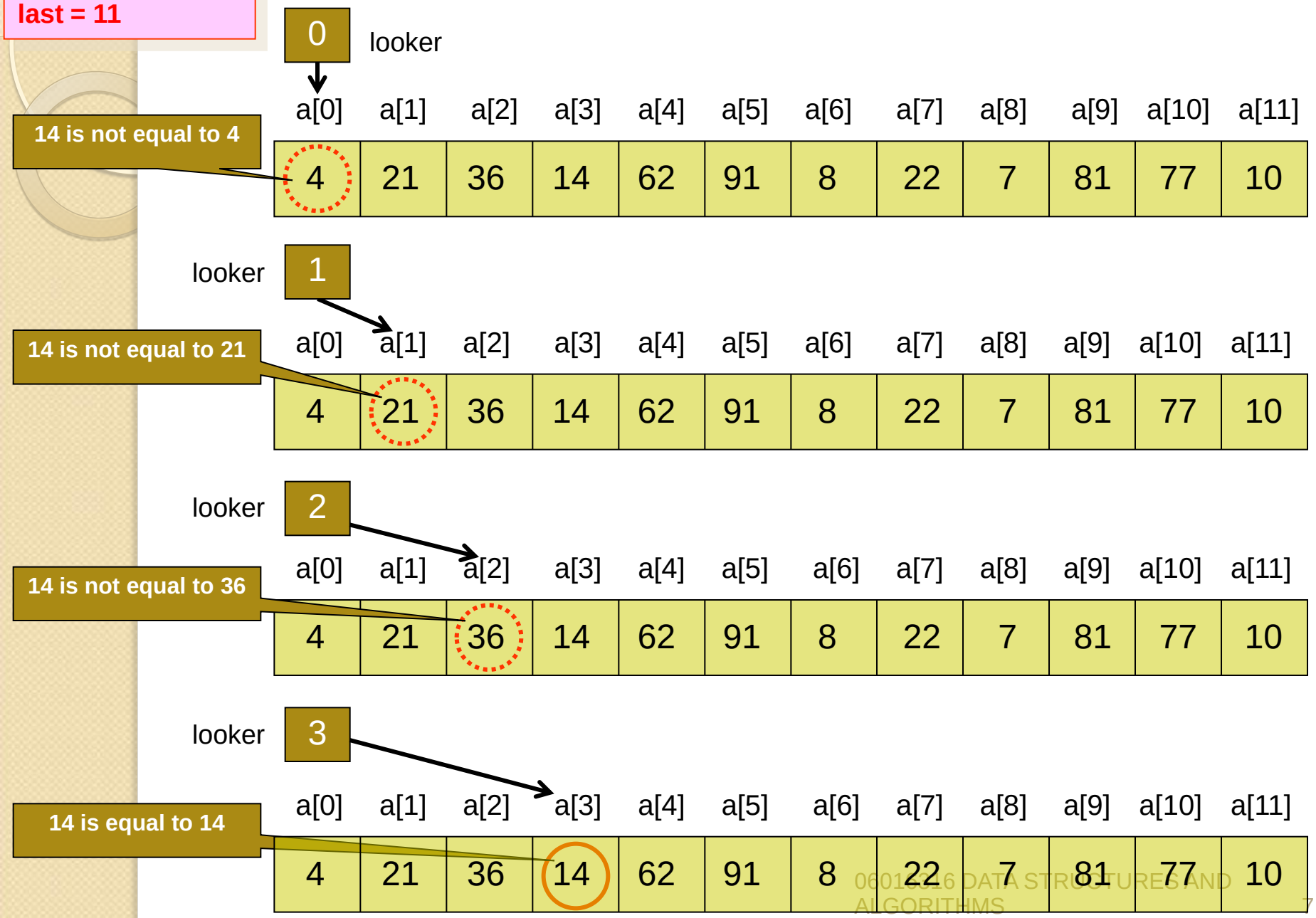


Sequential Search Algorithm in an Unordered List

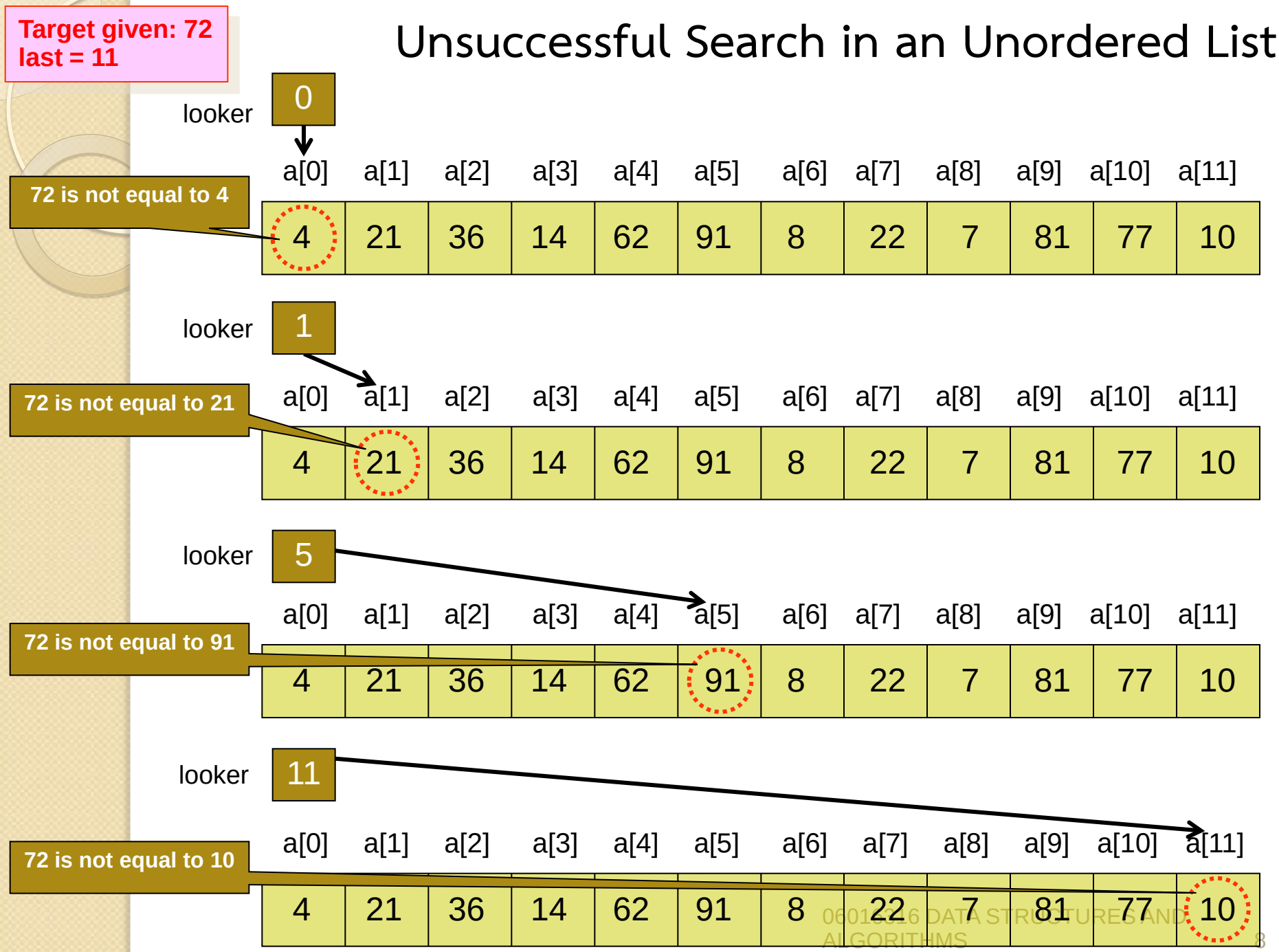
```
Algorithm seqSearch1(list, last, target)
1       set looker to 0
2       loop (looker < last AND target not equal list[looker])
    1       increment looker
3       end loop
4       set loc to looker
5       if (target equal list[looker])
    1       set found to loc
6       else
    1       set found to false
7       end if
8       return found
End seqSearch1
```

Successful Search of an Unordered List

Target given: 14
Location found: 3
last = 11



Unsuccessful Search in an Unordered List



Sequential Search Algorithm in an Ordered List

```
Algorithm seqSearch2(list, last, target)
1       set looker to 0
2       loop (looker < last AND loop is TRUE)
    1       if (target is equal to list[looker])
        1       set loc to looker
        2       set loop to FALSE
    2       else
        1       if (list[looker] > target)
            1       set loop to FALSE
        2       else
            1       increment looker
3       end loop
4       return loc
End seqSearch2
```

Search Algorithm Analysis

- วิเคราะห์ประสิทธิภาพของการค้นหา พิจารณาแบ่งเป็น 3 กรณี
  - กรณีดีที่สุด (Best Case)
  - กรณีแย่ที่สุด (Worst Case)
  - กรณีเฉลี่ย (Average Case)

Sequential Search Algorithm Analysis

- กรณีดีที่สุด สามารถค้นพบข้อมูลที่ต้องการได้ทันที (พบข้อมูลตั้งแต่แรก)
  - ทำการเปรียบเทียบแค่ 1 ครั้ง
- กรณีแย่ที่สุด ต้องทำการเปรียบเทียบข้อมูลไปจนถึงตัวสุดท้ายจึงจะพบ
  - ทำการเปรียบเทียบ N ครั้ง (มีข้อมูลทั้งหมด N ข้อมูล)
- กรณีเฉลี่ย หาโดยนำผลรวมของการเปรียบเทียบไปยังข้อมูลแต่ละตัว แล้วหารด้วยจำนวนข้อมูล
  - ทำการเปรียบเทียบเฉลี่ย (1+2+...+N)/N = (N+1)/2 ครั้ง

The Big-O Notation

- การวิเคราะห์ประสิทธิภาพของการค้นหาข้อมูล จะเน้นที่ชุดข้อมูลที่มีจำนวนมากๆ ซึ่งมีผลกระทบต่อเวลาในการทำงานโดยรวมมากขึ้น
- การวัดประสิทธิภาพในการค้นหาข้อมูล จะใช้ค่าประมาณซึ่งพิจารณาจากค่าที่มีผลกระทบมากที่สุด ตามแนวคิดของบิ๊กโอ (Big O)
- เช่น $f(N) = N^4 + 10N - 5$, $f(n) = O(N^4)$

Big-O of Sequential Search

- Best Case
  - เปรียบเทียบ 1 ครั้ง
  - O(1)
- Average Case
  - เปรียบเทียบเฉลี่ย (N+1)/2
  - O(N)
- Worst Case
  - เปรียบเทียบ N ครั้ง
  - O(N)

Binary Search

- การค้นหาแบบ Sequential ง่าย แต่เสียเวลามาก (ลองนึกถึงกรณีที่มีข้อมูลมากๆ)
- การค้นหาที่มีประสิทธิภาพมากขึ้น ได้แก่ การค้นหาแบบ Binary Search
- เป็นการค้นหาที่***ใช้กับชุดข้อมูลที่มีการเรียงลำดับเท่านั้น***

Binary Search (cont.)

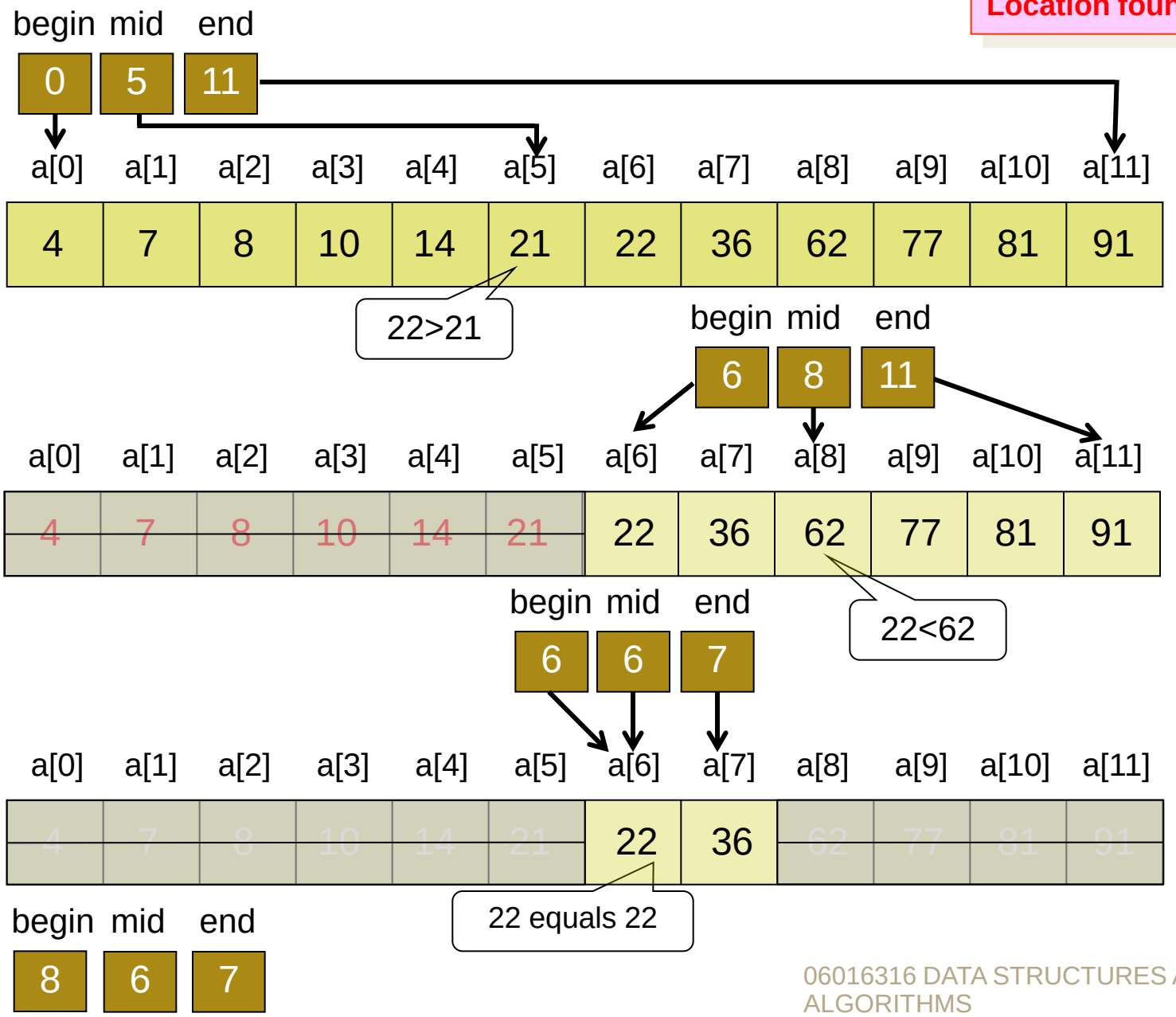
- ทำการเปรียบเทียบข้อมูลที่ต้องการกับข้อมูลที่อยู่ตำแหน่งกึ่งกลางของชุดข้อมูลทั้งหมด
- ถ้าไม่เท่ากัน (ไม่พบ) ให้ไล่เปรียบเทียบไปเรื่อยๆ จนกว่าจะพบหรือไม่สามารถแบ่งครึ่งชุดข้อมูลได้อีก โดยกำหนดว่าหาก
  - ค่าที่ต้องการค้นหาน้อยกว่า ให้เปรียบเทียบกับจุดกึ่งกลางของข้อมูลครึ่งแรกของชุดข้อมูลที่พิจารณา และไม่สนใจข้อมูลครึ่งหลังอีก
  - ค่าที่ต้องการค้นหามากกว่า ให้เปรียบเทียบกับจุดกึ่งกลางของข้อมูลครึ่งหลังของชุดข้อมูลที่พิจารณา และไม่สนใจข้อมูลครึ่งแรกอีก
- ตำแหน่งกึ่งกลาง : $\lfloor (begin + end)/2 \rfloor$

Binary Search Algorithm

```
Algorithm binarySearch(list, last, target)
1       set begin to 0
2       set end to last
3       loop (begin <= end)
    1       set mid to (begin+end)/2
    2       if (target > list[mid])
        1       set begin to mid+1
    3       else if (target < list[mid])
        1       set end to mid-1
    4       else
        1       set begin to (end+1)
    5       end if
4       end loop
5       set loc to mid
6       if (target is equal to list[mid])
    1       set found to TRUE
7       else
    1       set found to FALSE
8       end if
9       return found
End binarySearch
```

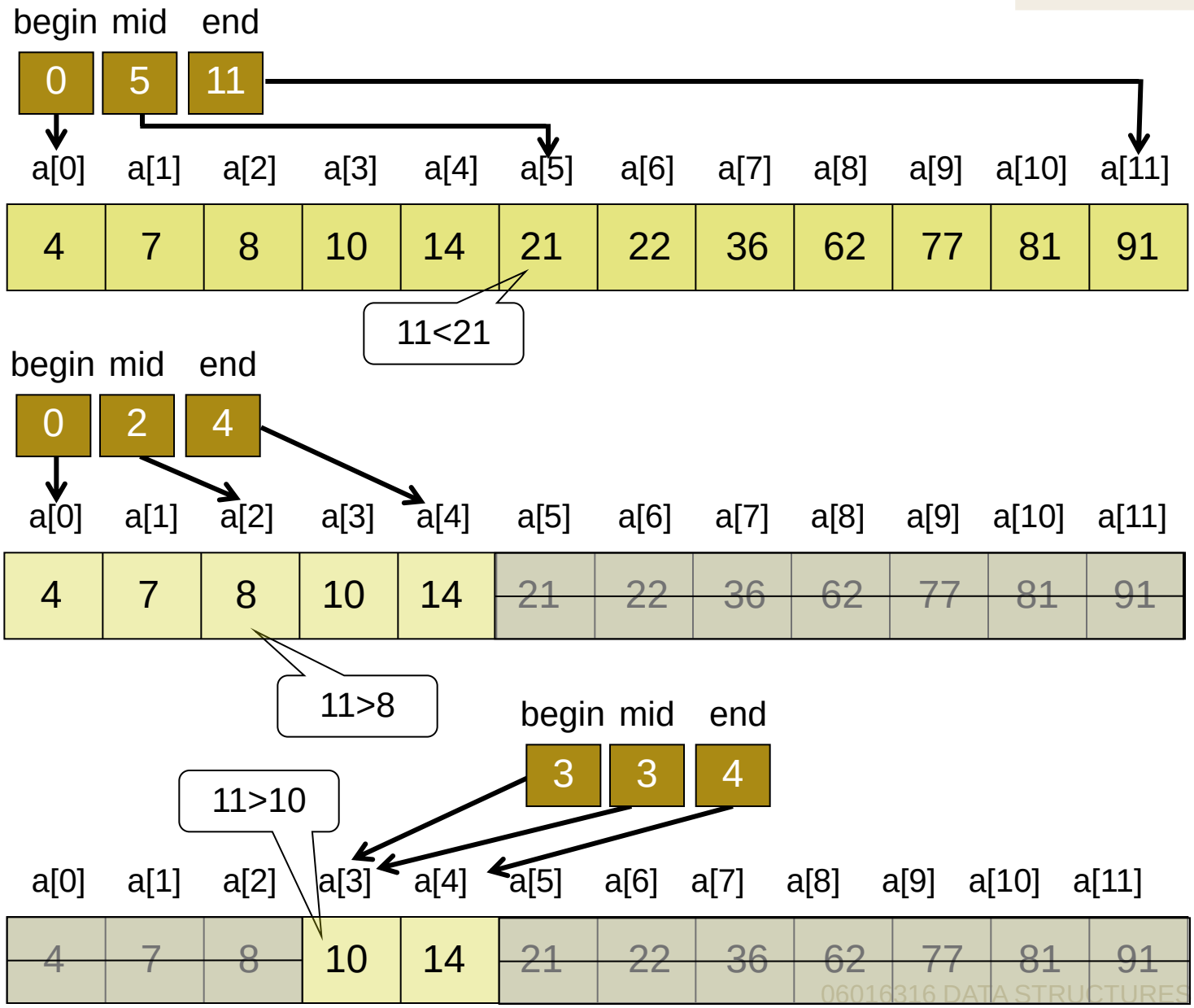
Successful Binary Search Example

Target given: 22
Location found: 6



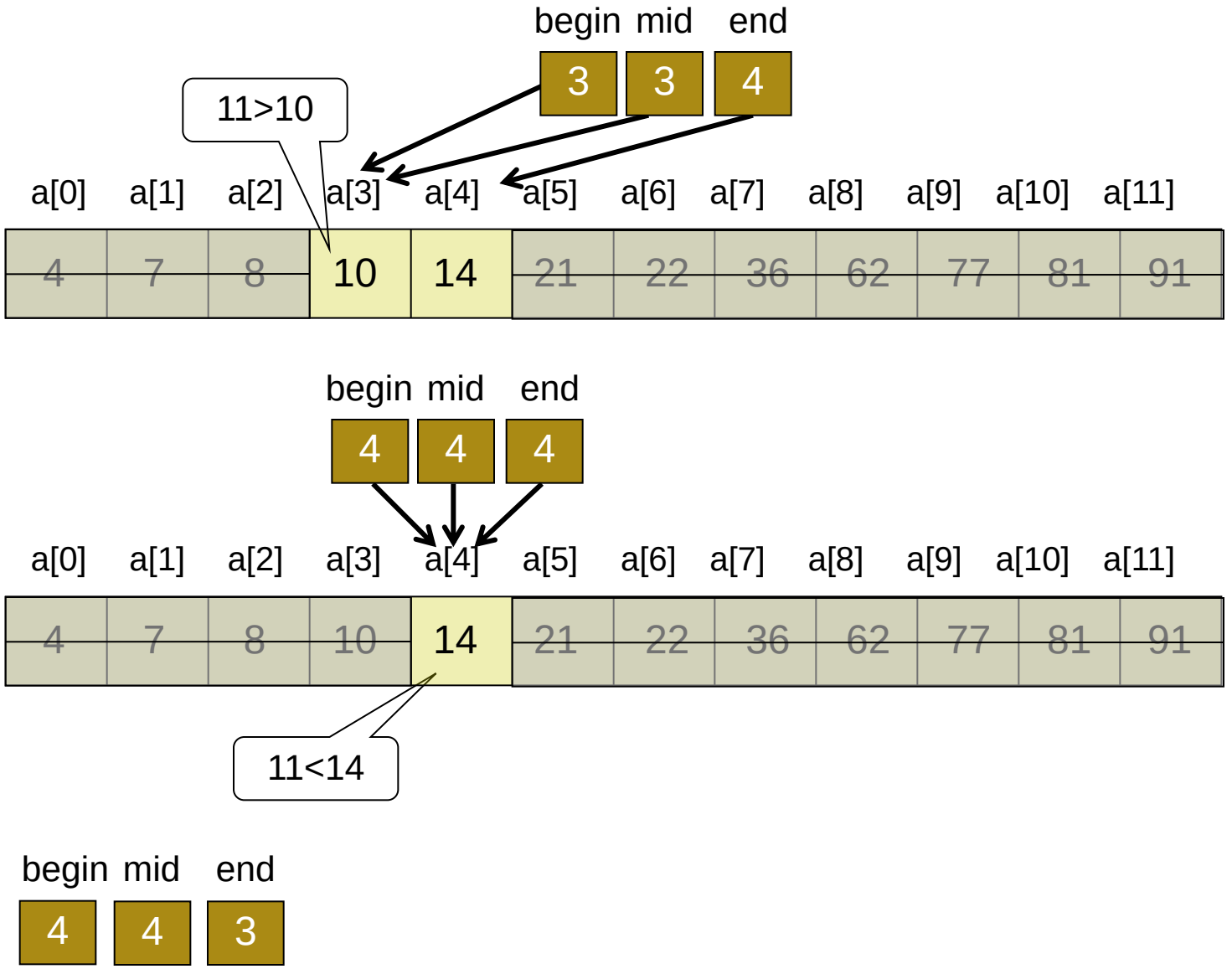
Unsuccessful Binary Search Example

Target given: 11



Unsuccessful Binary Search Example (cont.)

Target given: 11



Big-O of Binary Search

- Best Case
  - เปรียบเทียบ 1 ครั้ง ข้อมูลที่ต้องการอยู่กึ่งกลางพอดี
  - O(1)
- Average Case
  - เปรียบเทียบเฉลี่ย $(2^0*1+2^1*2+2^2*3+...+2^m*(m+1))/N$

    (m = ระดับของต้นไม้)
  - $O(\log_2 N)$
- Worst Case
  - เปรียบเทียบโดยแบ่งข้อมูลออกเป็นชุดย่อยๆ จนเหลือตัวเดียว
  - $O(\log_2 N)$

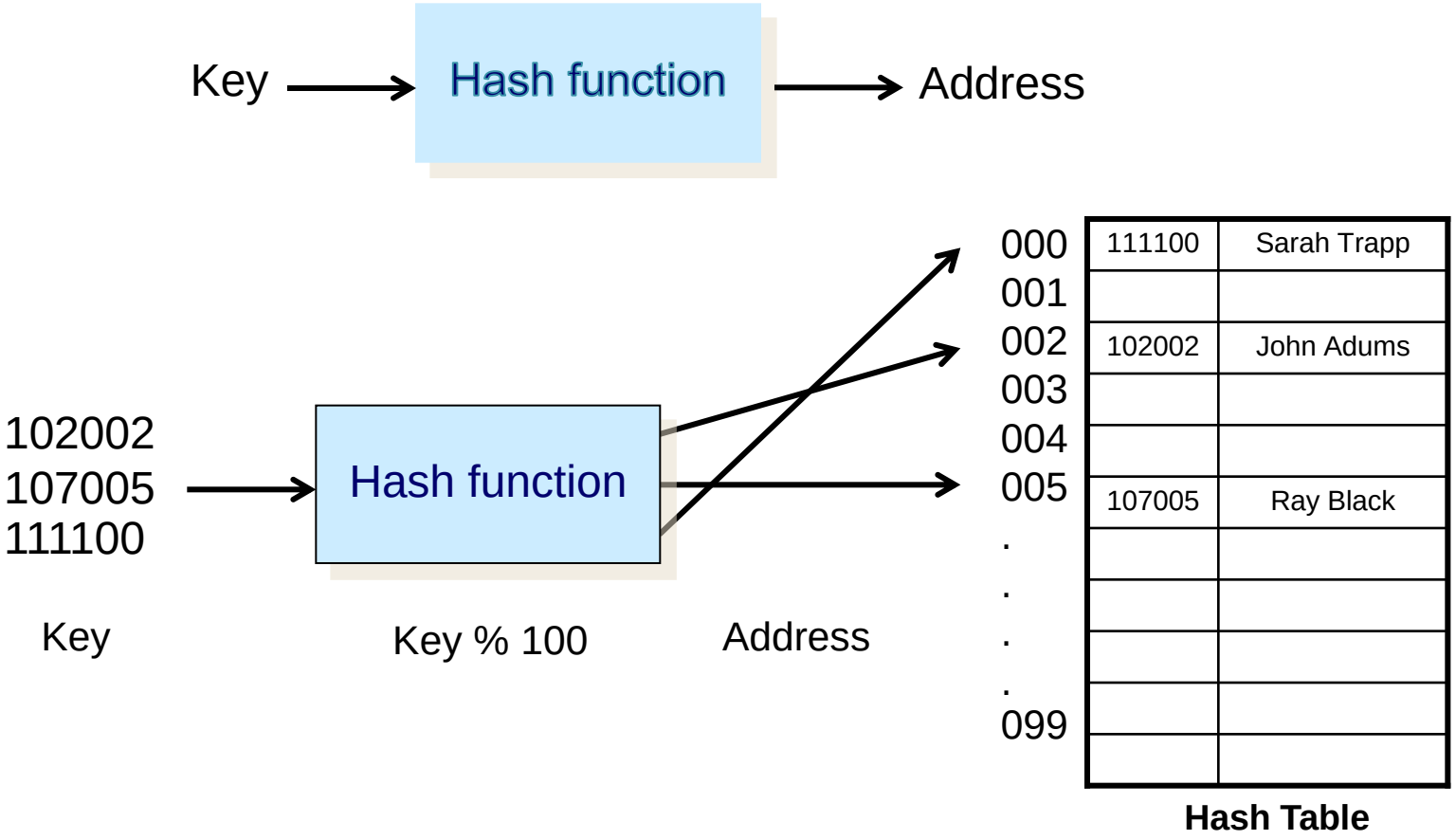
Comparison of Binary and Sequential Searches

Element size	Iterations	
	Binary	Sequential
16	4	16
50	6	50
256	8	256
1,000	10	1,000
10,000	14	10,000
100,000	17	100,000
1,000,000	20	1,000,000

Hashing

- ต้องการให้การค้นหาข้อมูลมีประสิทธิภาพเป็น O(1) นั่นคือ สามารถเข้าถึงข้อมูลได้โดยตรง
- ***หลักการ***
  - ข้อมูลที่จะนำเข้า หรือค้นหา จะเรียกว่า คีย์ (key)
  - ใช้ Hash Function หาตำแหน่งของคีย์นั้นที่เก็บข้อมูล ซึ่งเรียกว่า ตารางแฮช (Hash Table)

Hashing (cont.)



Collision

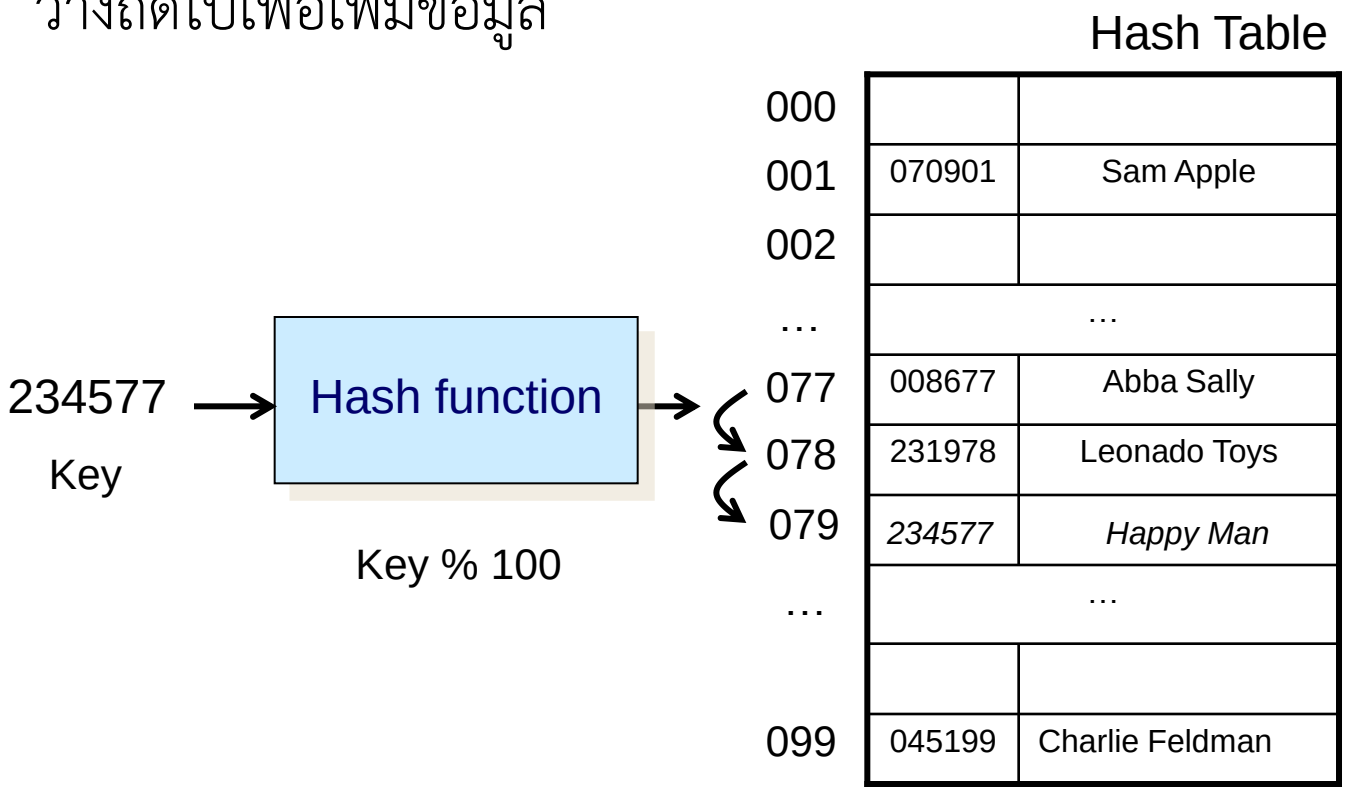
- ***ปัญหา*** อาจเกิดการชนกันของข้อมูล นั่นคือ คีย์ไม่เหมือนกันแต่แฮชแล้วได้ค่าเดียวกันในตารางแฮช
- เช่น กำหนด Hash Function เป็นการ mod ด้วย 100 และมีคีย์เป็น 10275 และ 27675 เมื่อผ่านการแฮชแล้วจะได้ 75 เหมือนกัน -> เกิดการชนกัน
- การหลีกเลี่ยงการชนกัน (Collision)
  - กำหนด Hash Function ให้ดี
  - นำคีย์ไปแฮชอีกครั้ง เรียกว่า Rehashing

Collision Resolution

- Linear Probing
- Buckets
- Chaining (Linked List Collision Resolution)

Linear Probing

- กรณีที่นำข้อมูลเข้า เมื่อแฮชแล้วเกิดการชนกัน ให้หาตำแหน่งที่ว่างถัดไปเพื่อเพิ่มข้อมูล



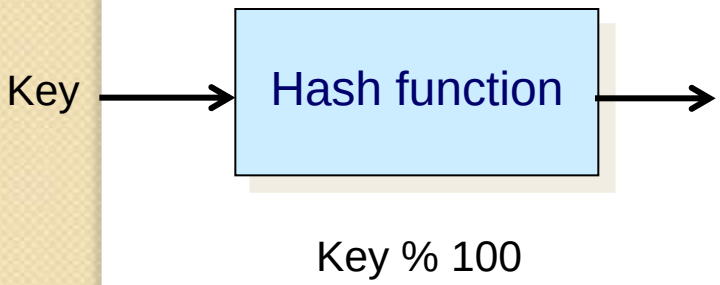
Linear Probing

- ในกรณีที่ต้องการค้นหาข้อมูล
  - นำคีย์ไปผ่าน Hash Function ถ้าเจอข้อมูลที่มีคีย์ตรงกันแสดงว่าหาเจอ
  - ถ้าไม่ใช่คีย์ที่ต้องการ ให้ไล่ไปตำแหน่งถัดไปเรื่อยๆจนกว่าจะเจอ
  - แต่ถ้าเจอช่องว่าง แสดงว่า ไม่มีข้อมูลนั้น
- เมื่อเกิดการชนกัน จะไล่ไปตำแหน่งถัดไป นั่นก็คือ การทำ Rehashing ด้วยฟังก์ชัน (ตำแหน่งที่แฮชได้ + ค่าคงที่) % ขนาดอาร์เรย์
- การ Hashing เป็นวิธีที่ดี แต่เมื่อเกิดการชนกันมากขึ้น จะทำให้ประสิทธิภาพการค้นหาข้อมูลแย่ลง

Buckets

- คีย์ที่ผ่านการแฮชแล้วได้ค่าเหมือนกัน สามารถเก็บอยู่ในตำแหน่งเดียวกันได้ ซึ่งมีการกำหนดจำนวนของคีย์ที่ตำแหน่งเดียวกันชัดเจน
- เช่น กำหนดให้มีคีย์ที่ตำแหน่งเดียวกันไม่เกิน 3 คีย์
- เก็บข้อมูลตารางแฮชเป็นแบบอาร์เรย์ 2 มิติ
- ***ปัญหา*** หากมีคีย์ที่แฮชแล้วจะอยู่ตำแหน่งเดียวกันมากกว่าที่กำหนด ก็จะเกิดการชนกันอีก

Buckets (cont.)



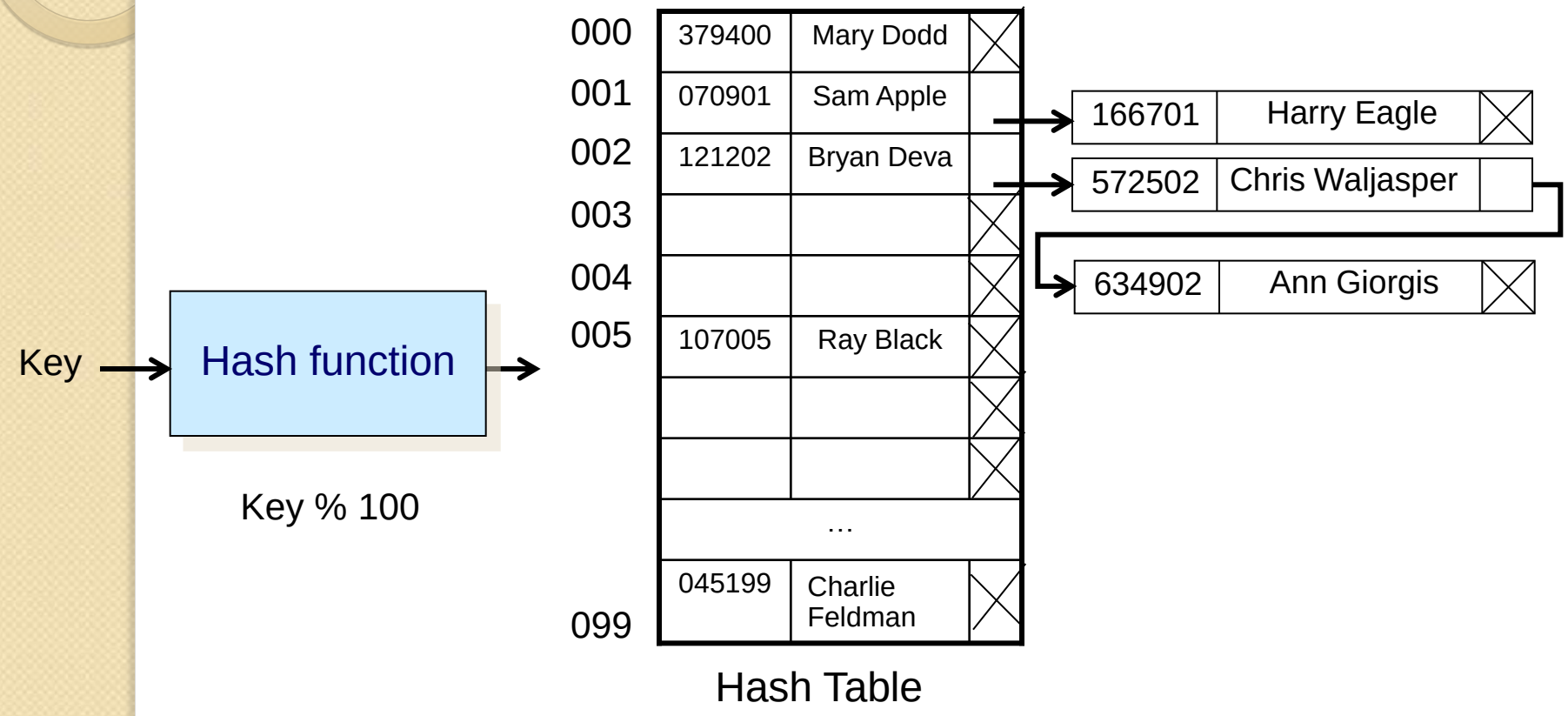
000	Bucket 0	379400	Mary Dodd
001	Bucket 1	070901	Sam Apple
		166701	Harry Eagle
002	Bucket 2	121202	Bryan Devaux
		572502	Chris Waljasper
		634902	Ann Giorgis
…			
099	Bucket 99	045199	Charlie Feldman

Hash Table

Chaining

- คีย์ที่ผ่านการแฮชแล้วได้ค่าเหมือนกัน สามารถเก็บอยู่ในตำแหน่งเดียวกันได้ ***โดยไม่จำกัดจำนวน***
- เก็บข้อมูลตารางแฮชเป็นแบบลิงค์ลิสต์

Chaining (cont.)



Quiz

- หากกำหนดให้อาร์เรย์ มีค่าข้อมูลดังนี้

  18 13 17 26 44 56 88 97

  ให้ใช้วิธี Sequential Search และ Binary Search ในการค้นหา

  - ข้อมูล 56
  - ข้อมูล 20

  (ให้เขียนการค้นหาเป็นลำดับขั้นตอน และสรุปด้วยว่าต้องมีการเปรียบเทียบกี่ครั้งจึงจะพบข้อมูลที่ต้องการ)

Quiz

- ให้เก็บข้อมูลเหล่านี้ลงในตารางแฮชขนาด 20 ช่อง โดยใช้วิธี Linear Probing
  - กำหนดให้ใช้ Hash Function แบบ Modulo-division

224562	137456	214562
140145	214576	162145
144467	199645	234534

- การทำงานดังกล่าว ทำให้เกิดการชนกันของข้อมูลกี่ครั้ง อย่างไร