



# การอบรมคอมพิวเตอร์โอลิมปิกค่าย 2

NON-LINEAR DATA STRUCTURES -- BINARY HEAP, GRAPH, HASHING



## จัดลำดับงานที่ต้องทำ

ทำ  
การบ้าน

ทำความ  
สะอาด  
บ้าน

ล้างจาน

ออนไลน์  
ช้อปปิ้ง

อ่าน  
หนังสือ

เล่นเกม

จัดลำดับงานที่ต้องทำ

1

5

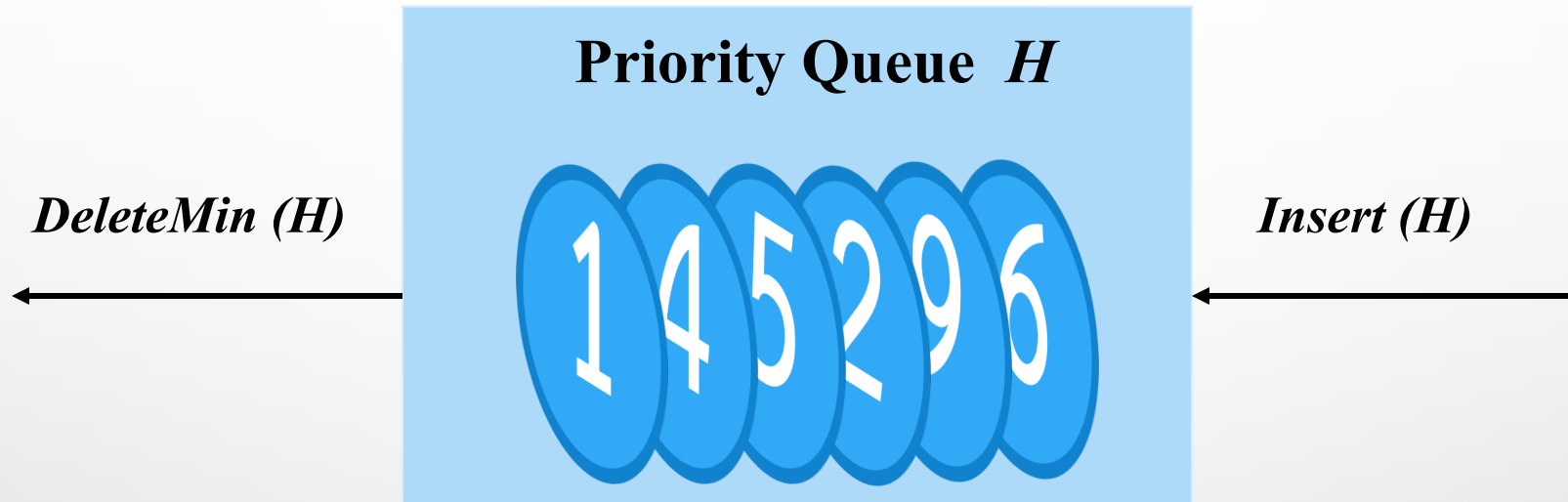
9

4

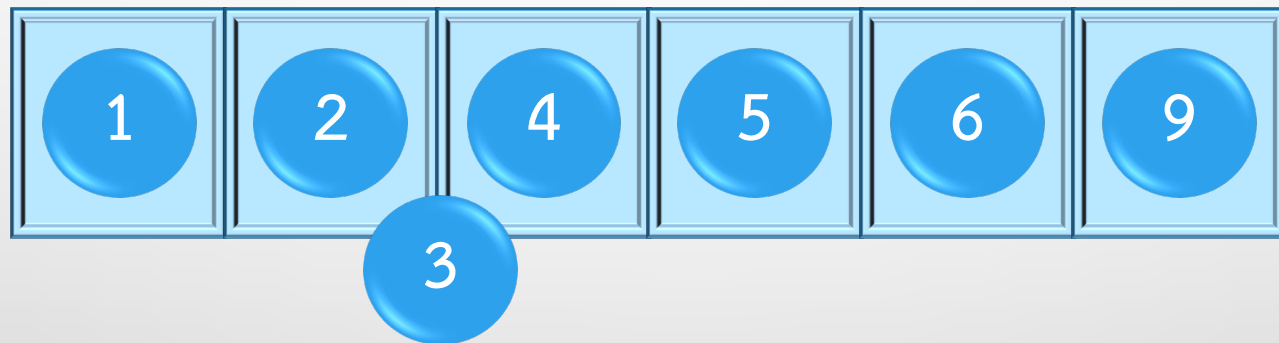
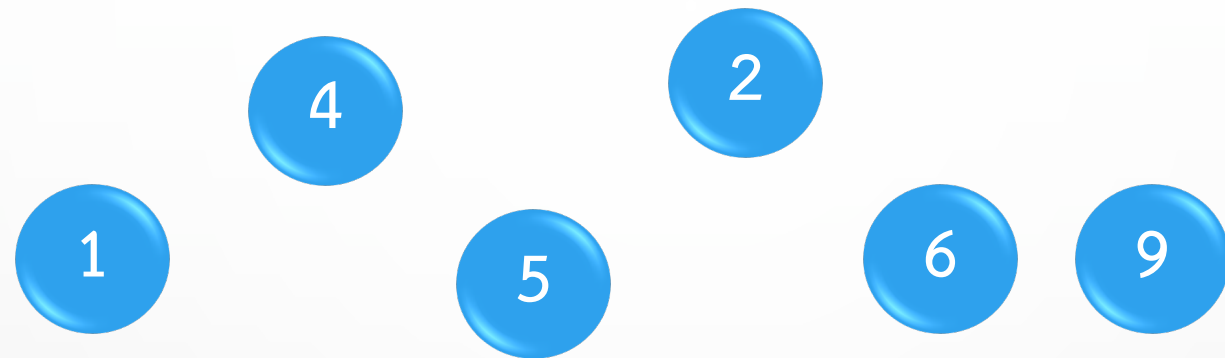
2

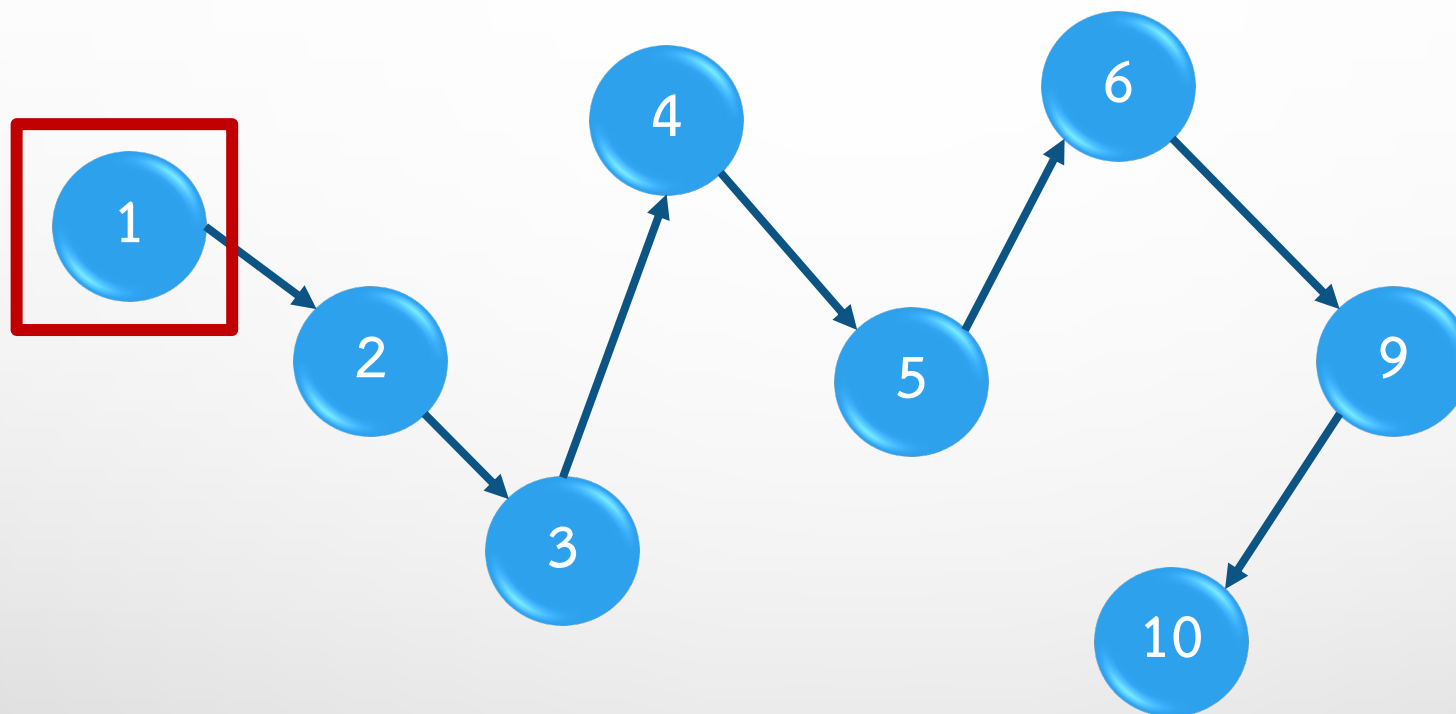
6

# PRIORITY QUEUES (BINARY HEAPS)



- **PRIORITY QUEUE** คือโครงสร้างข้อมูลที่ประกอบไปด้วย 2 OPERATIONS หลัก คือ การแทรกข้อมูล (INSERT) และการลบข้อมูลตัวที่น้อยที่สุด (DELETEMIN)
- โครงสร้างข้อมูลที่สามารถสร้าง PRIORITY QUEUE เช่น ลิงค์ลิสต์ อะเรย์ หรือต้นไม้ไบนารี ที่เรียกว่า BINARY HEAPS





# BINARY HEAP

# คุณสมบัติของ BINARY HEAPS

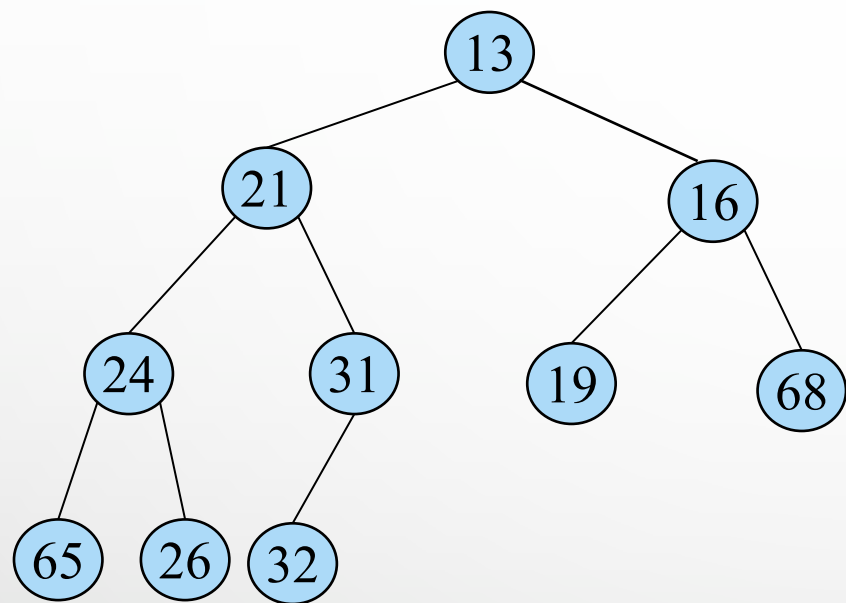
## คุณสมบัติทางด้านโครงสร้าง

- BINARY HEAPS เป็นต้นไม้ไบนารีที่โหนดถูกเติมเต็มลงมาที่ระดับจากซ้ายไปขวา หรือเป็นต้นไม้ไบนารีที่เกือบสมบูรณ์

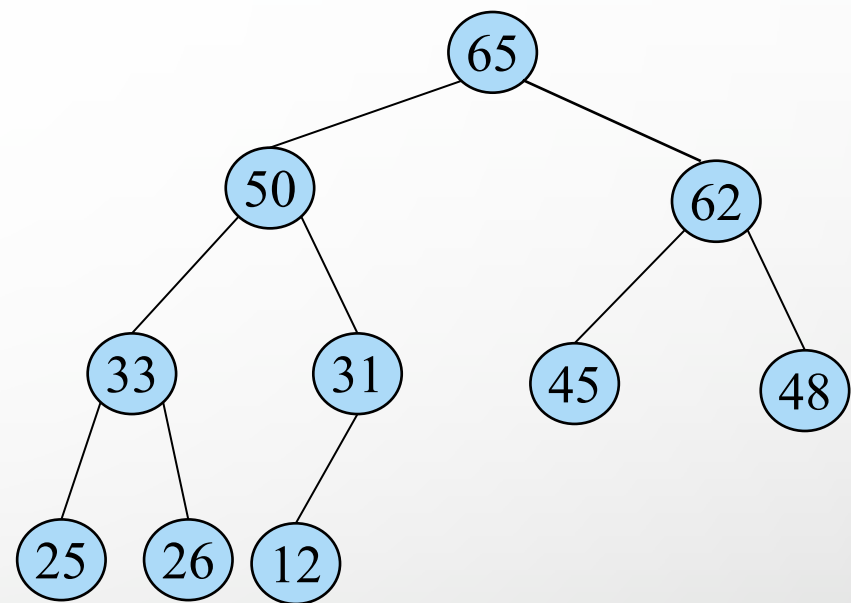
## ประเภทของ BINARY HEAP

- MAX HEAP: KEY OF EACH NODE IS ALWAYS GREATER THAN ITS CHILD NODE/S AND THE KEY OF THE ROOT NODE IS THE LARGEST AMONG ALL OTHER NODES;
- MIN HEAP: KEY OF EACH NODE IS ALWAYS SMALLER THAN THE CHILD NODE/S AND THE KEY OF THE ROOT NODE IS THE SMALLEST AMONG ALL OTHER NODES.:





**Min-Heap**



**Max-Heap**


# MIN-HEAP

## คุณสมบัติทางด้านลำดับของข้อมูล

- ข้อมูลใน HEAPS จะถูกจัดเรียงเพื่อให้การลบข้อมูลตัวที่น้อยที่สุดออกได้สะดวก ง่ายและรวดเร็ว
- เพื่อให้การค้นหาข้อมูลตัวที่เล็กที่สุดทำได้เร็ว ข้อมูลตัวที่เล็กที่สุด จะอยู่ที่โหนดรากหรือ ROOT NODE
- ต้นไม้ย่อยต้องมีคุณสมบัติเป็น HEAPS ด้วยคือทุกๆ โหนดจะต้องมีค่าน้อยกว่าโหนดลูกโหนดหลานของตัวเอง



# OPERATION

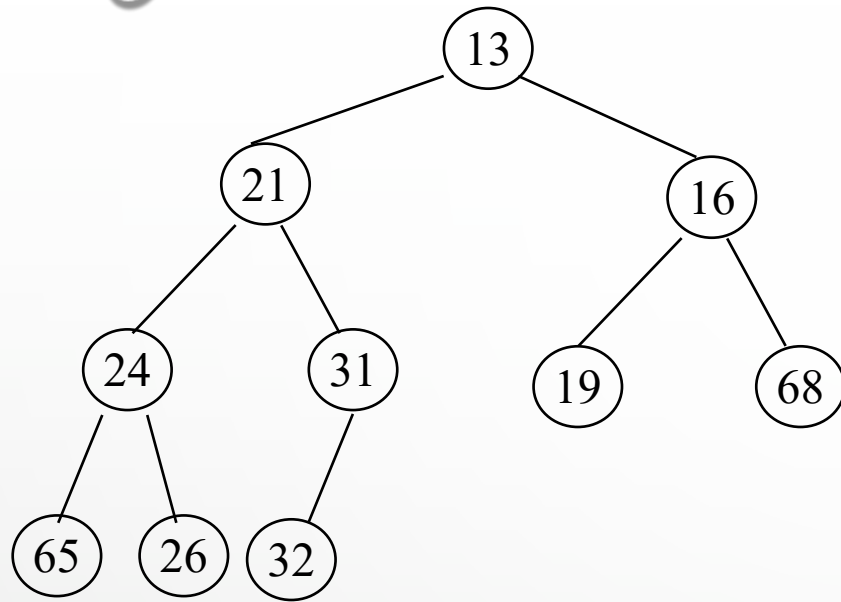
- HEAPIFY
  - INSERTION
  - DELETETION
- 

## IMPLEMENTATION BINARY HEAPS WITH ARRAY

- การเก็บข้อมูลในไบนารีเหมาะกับการใช้อะเรย์ แบบมีลำดับ (SEQUENTIAL ARRAY REPRESENTATION) เนื่องจากสามารถใช้เนื้อที่ได้เต็มประสิทธิภาพ

# โครงสร้างของ HEAP NODE

```
HEAP    <ARRAY>           // ตัวแปรที่ใช้เก็บข้อมูล  
SIZE    <INTEGER>         // เก็บขนาดของ HEAP
```



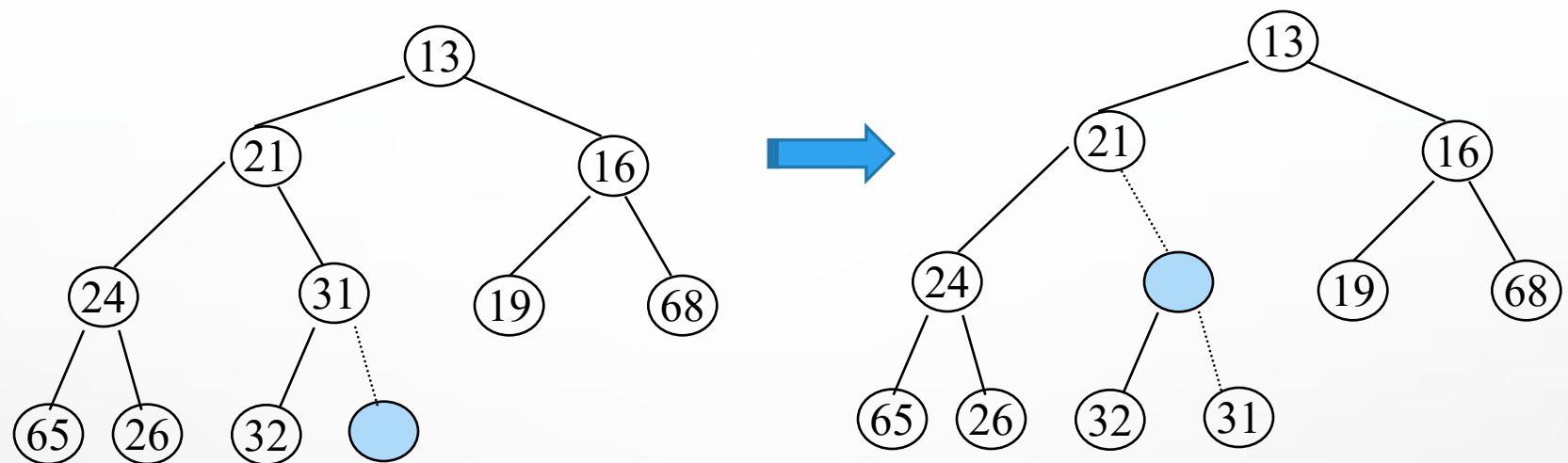
A heap in its logical form

โหนด  $i$  ใดๆ จะมีลูกทางซ้ายอยู่  
ตำแหน่ง  $2i+1$  ในอาร์เรย์ และลูก  
ทางขวาอยู่ที่ตำแหน่ง  $2i+2$

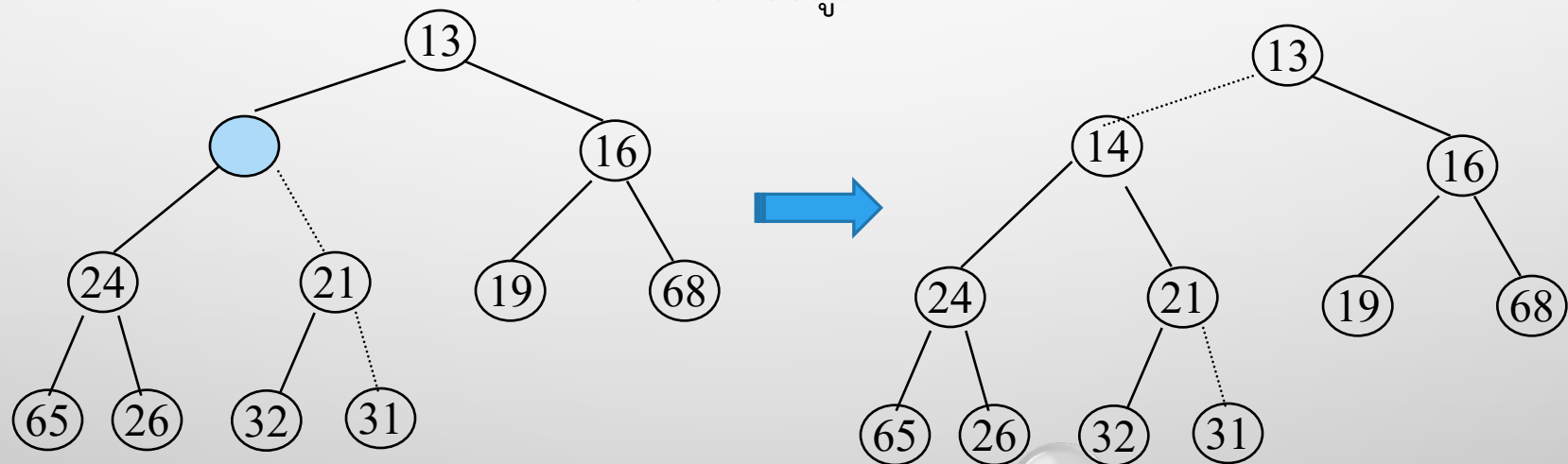
0	1	2	3	4	5	6	7	8	9	10	11	12
13	21	16	24	31	19	68	65	26	32			

A heap in an array

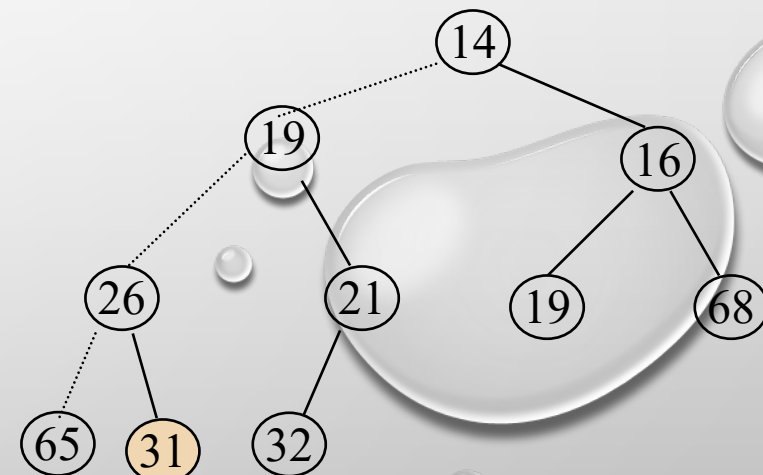
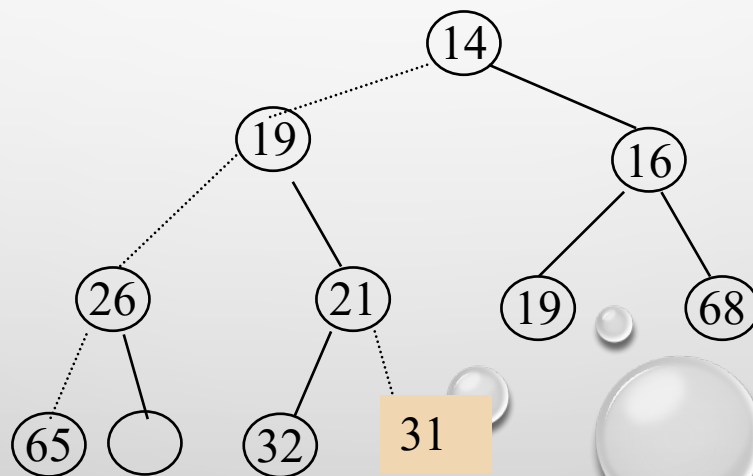
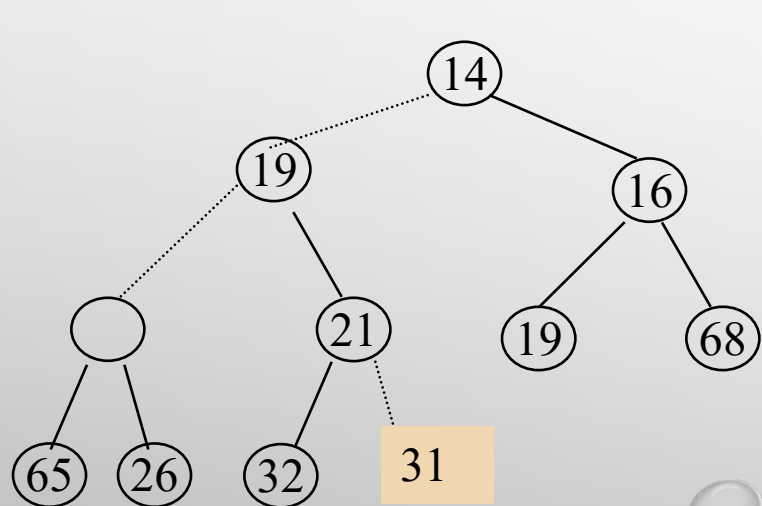
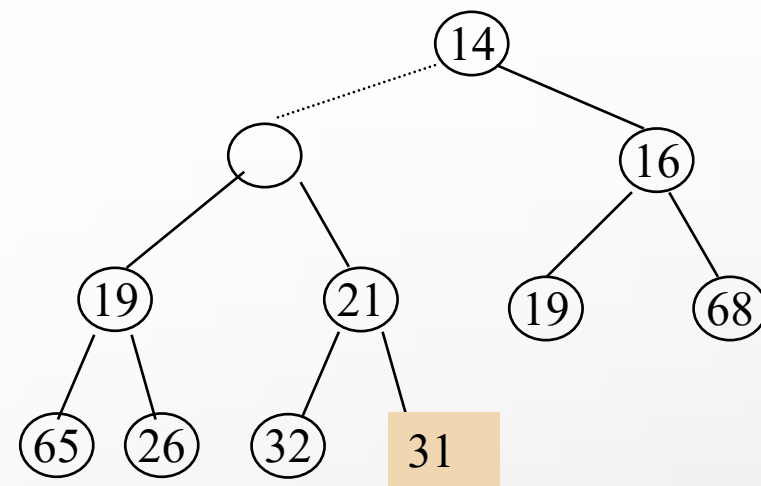
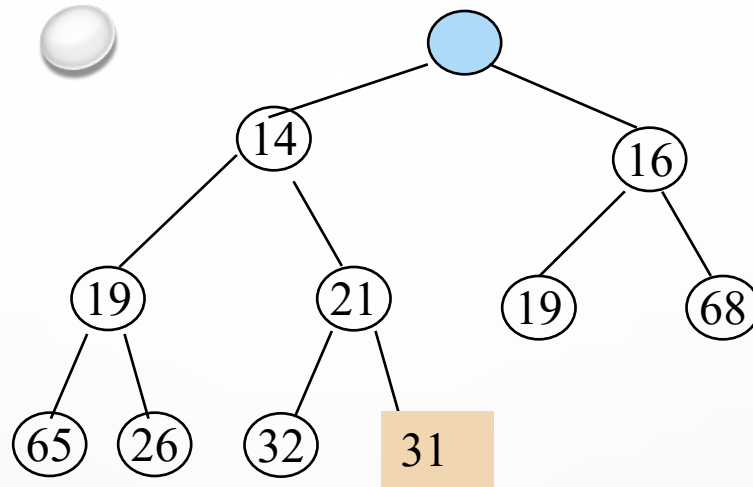
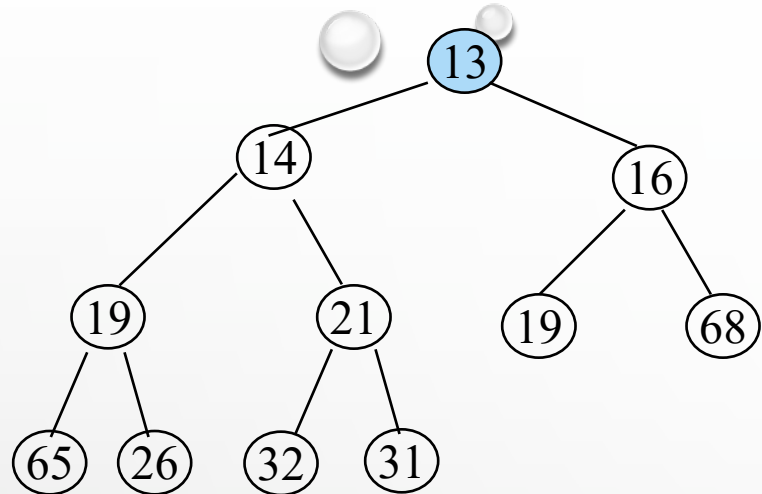
# INSERT ELEMENT INTO HEAP



การแทรกข้อมูล 14



# Delete Element from Heap





# HEAP ALGORITHMS

- **INSERTHEAP** แทรกโหนดใหม่ โดยเริ่มที่ตำแหน่งท้าย เปรียบเทียบ โหนดใหม่ กับ PARENT NODE ถ้าโหนดใหม่มีค่าน้อยกว่า จะมีการสลับค่ากับ PARENT NODE ทำไปเรื่อยๆ จนถึงตำแหน่งที่เหมาะสม
- **DELETEHEAP** ลบโหนดที่ ROOT เลือกโหนดเพื่อมาแทน โดยการเปรียบเทียบ CHILD NODE เพื่อหาค่าที่น้อยที่สุดลำดับถัดมาขึ้นมาแทน

**ALGORITHM** INSERTHEAP (VAL HEAP <ARRAY>, VAL SIZE <INTEGER>, VAL NEWVAL <KEY>)

GIVEN AN ARRAY, REARRANGE DATA SO THAT IT FORMS HEAP

**PRE**    HEAP IS ARRAY CONTAINING A HEAP

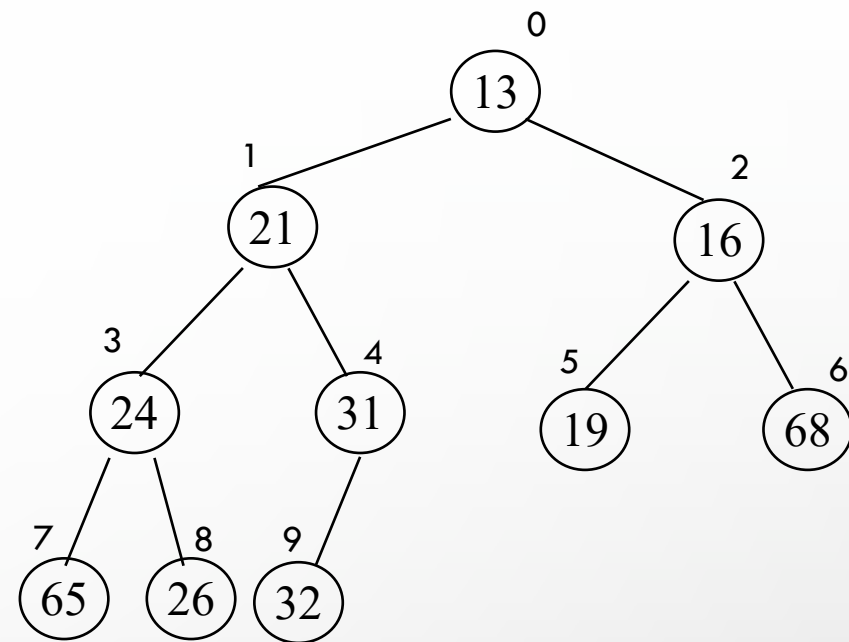
          SIZE IS THE SIZE OF HEAP

          NEWVAL IS NEW DATA TO BE INSERTED INTO HEAP

**POST**    NEWVAL HAS BEEN INSERTED INTO HEAP IN PROPER LOCATION

```
HOLE = SIZE
SIZE += 1
PARENT = (HOLE-1)/2
LOOP (HEAP[PARENT] >NEWVAL)
    HEAP[HOLE] = HEAP[PARENT]
    HOLE = PARENT
    PARENT = (HOLE-1)/2
HEAP[HOLE] = NEWVAL
RETURN
```

**END** INSERTHEAP



0	1	2	3	4	5	6	7	8	9	10	11	12
13	21	16	24	31	19	68	65	26	32			

**ALGORITHM** DELETEHEAP (VAL HEAP <ARRAY>, VAL SIZE <INTEGER>)

**DELETES ROOT OF HEAP AND PASSES DATA BACK TO CALLER**

**PRE**    HEAP IS ARRAY CONTAINING A HEAP AND SIZE IS HEAP SIZE

**RETURN**    RETURN THE KEY AT ROOT NODE

MINELEMENT = HEAP[0]

LASTELEMENT = HEAP[SIZE-1]

SIZE = SIZE - 1

HOLD = 0

**LOOP** (HOLD\*2+1 <= SIZE)

LEFTCHILD = HOLD \* 2 + 1

RIGHTCHILD = HOLD \* 2 + 2

IF (HEAP[RIGHTCHILD] >

HEAP[LEFTCHILD])

CHILD = HEAP[LEFTCHILD]

ELSE

CHILD = HEAP[RIGHTCHILD]

IF (LASTELEMENT >

HEAP[CHILD])

HEAP[HOLD] = HEAP[CHILD]

ELSE

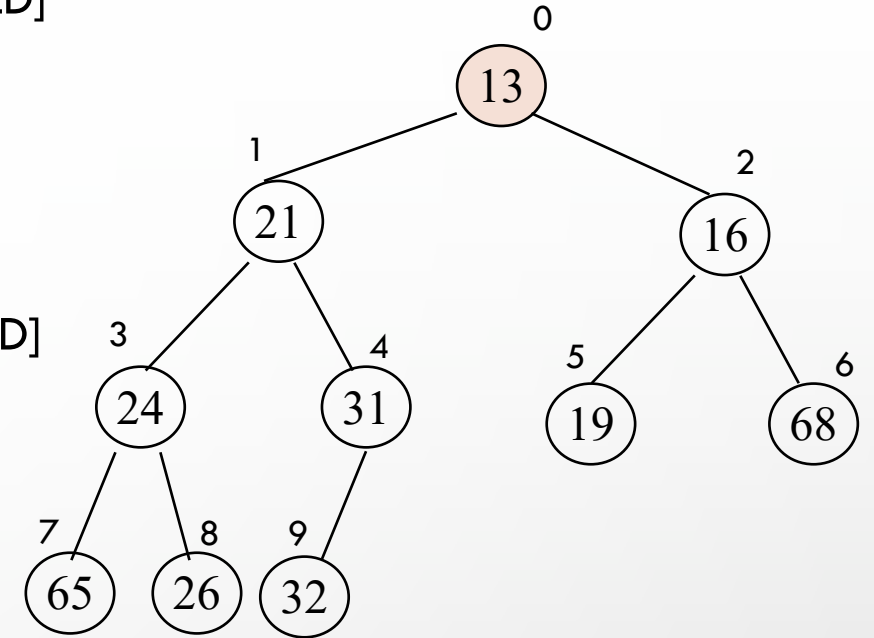
BREAK

**END LOOP**

HEAP[HOLD] = LASTELEMENT

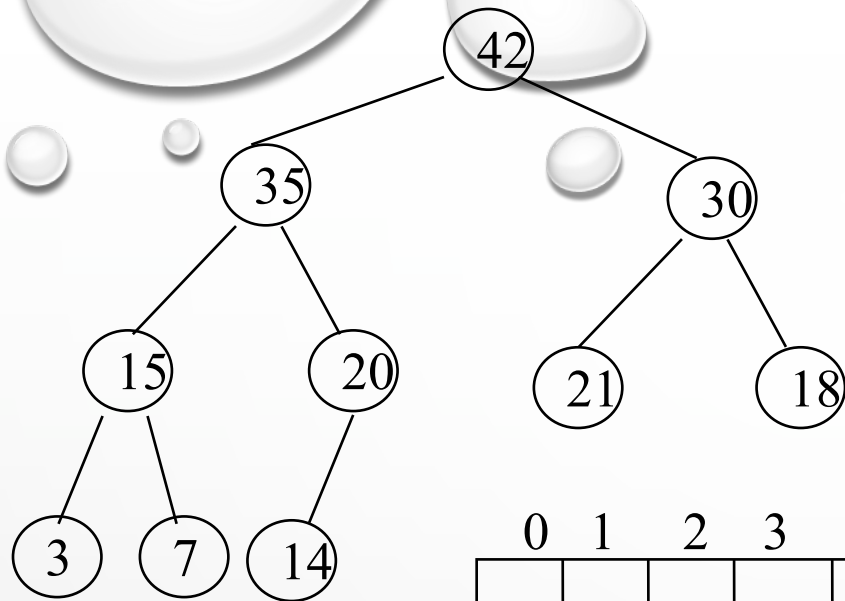
RETURN MINELEMENT

**END DELETEHEAP**

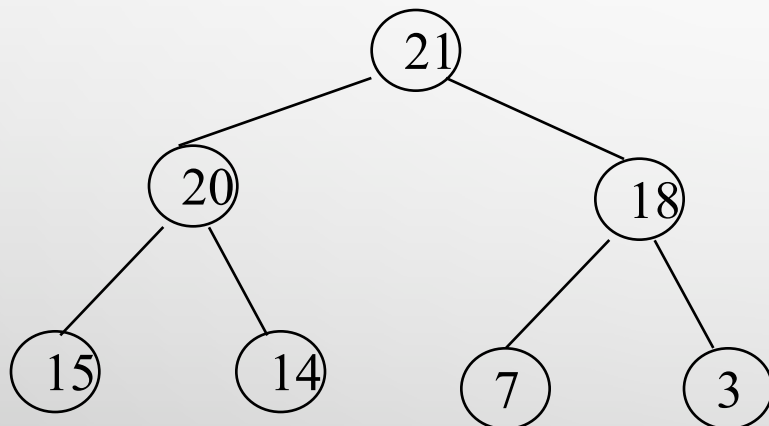


# การนำ HEAP ไปแก้ปัญหา

- การหาข้อมูลลำดับที่  $K$  ในลิสต์ของข้อมูลแบบไม่มีลำดับ
- วิธีการแก้ปัญหา
  - เรียงลำดับข้อมูลในลิสต์ แล้วเลือกข้อมูลลำดับที่  $K$
  - สร้าง HEAP และลบข้อมูลออก  $K-1$  จำนวน เหลือข้อมูลตัวที่ต้องการที่ตำแหน่ง ROOT แล้วค่อยแทรกข้อมูลตัวที่ลบออกกลับเข้าไปใน HEAP



0	1	2	3	4	5	6	7	8	9	10	11	12
42	35	30	15	20	21	18	3	7	14			



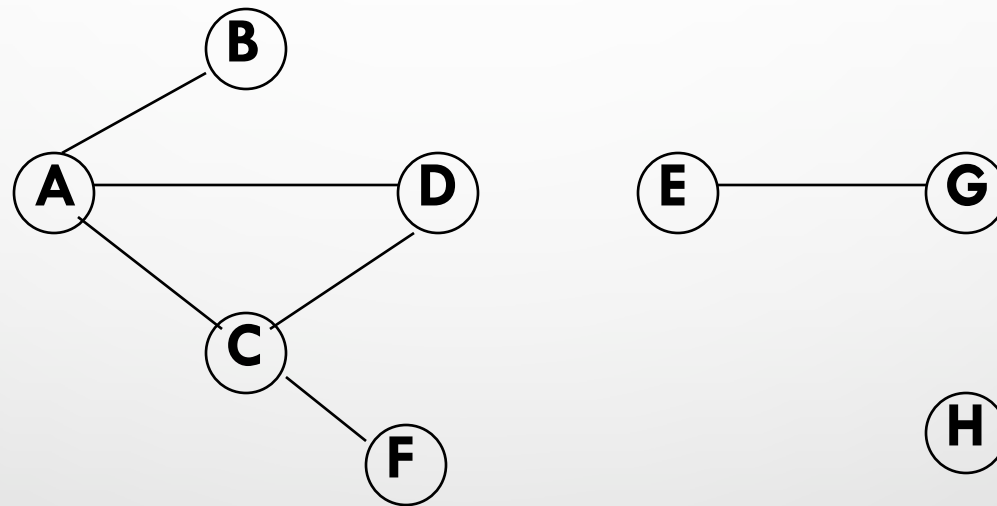
0	1	2	3	4	5	6	7	8	9	10	11	12
21	20	18	15	14	7	3	30	35	42			

# กราฟ (GRAPH)

NOTE : MOST SLIDES ARE FROM C++ CLASSES AND DATA STRUCTURES (JEFFREY CHILDS)

# กราฟ (GRAPH)

- **กราฟ** ประกอบไปด้วยกลุ่มของโนด (NODE) ที่เรียกว่า **VERTICES** และเส้นเชื่อมระหว่างโนดเรียกว่า **EDGES** ดังภาพ



เซตของ VERTICES คือ {A,B,C,D,E,F,G,H}

เซตของ EDGES คือ {(A,B),(A,D),(A,C),(C,D),(C,F),(E,G),(A,A)}

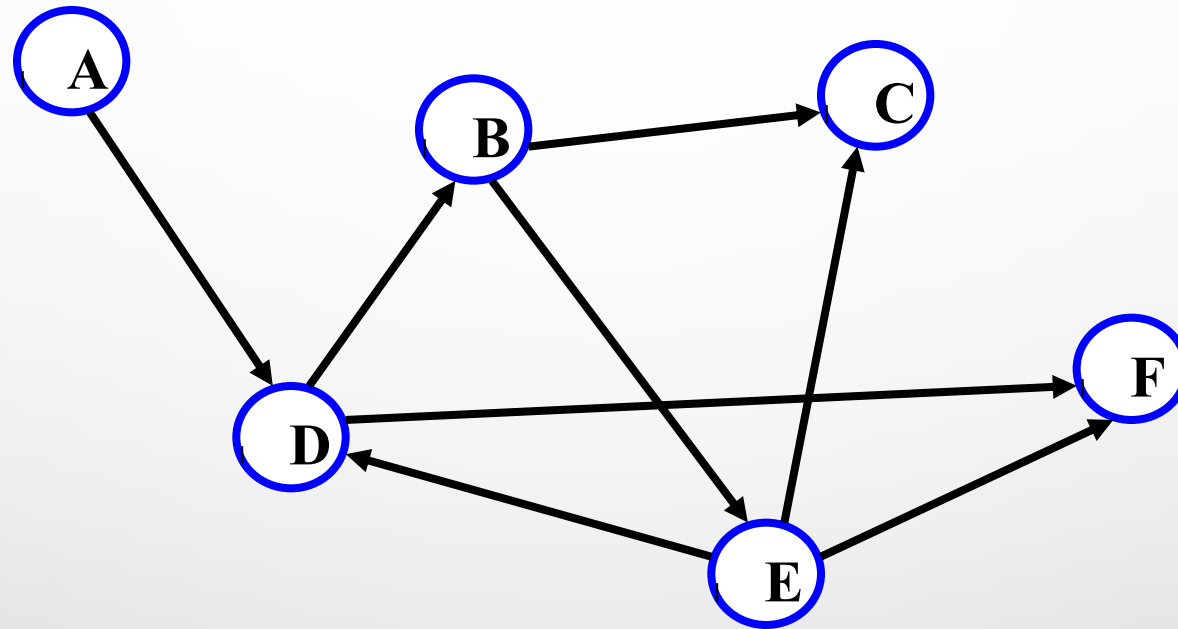


# กราฟ (Graph)

- กราฟไม่จำเป็นต้องเป็น tree แต่ tree เป็นกราฟ
- ไม่จำเป็นที่ทุก vertex ต้องมี edge มาเชื่อม
- vertex ในกราฟอาจจะชี้ไปยัง vertex อื่นๆ ในกราฟก็ได้ ไม่จำกัด
- เราจะกล่าวว่าโหนด  $n$  *incident* กับ edge  $x$  เมื่อ  $n$  เป็นหนึ่งในสองโหนดที่เชื่อมกับ edge  $x$
- *degree* ของโหนด หมายถึงจำนวน edge ที่เชื่อมเข้ามาที่โหนดนั้น

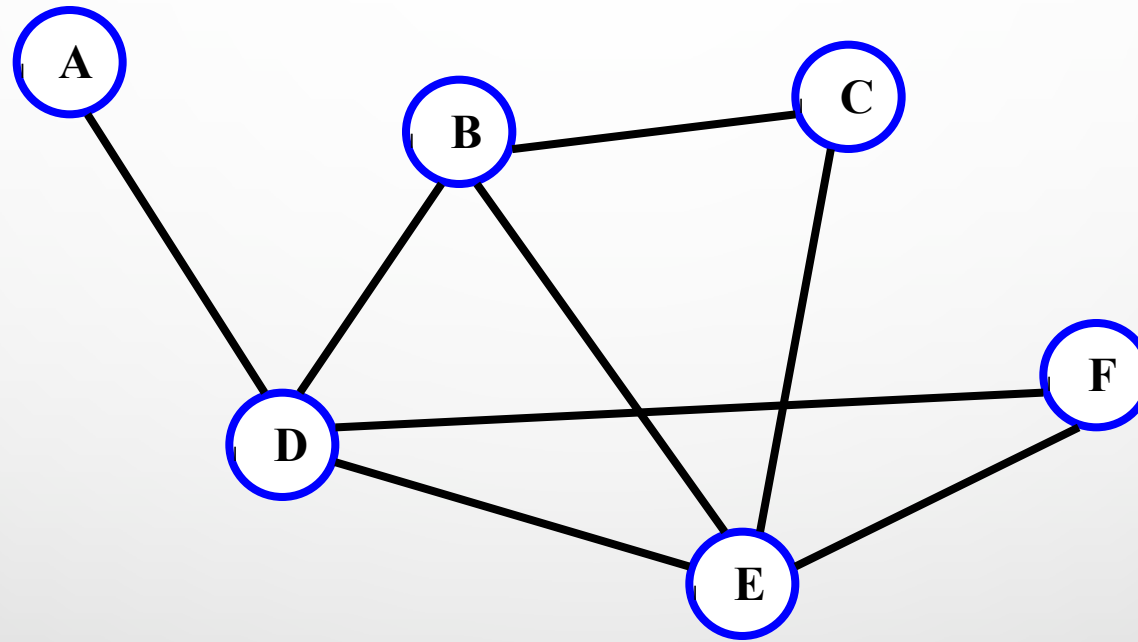
# กราฟแบบมีทิศทาง

(A Directed Graph หรือ Digraph)



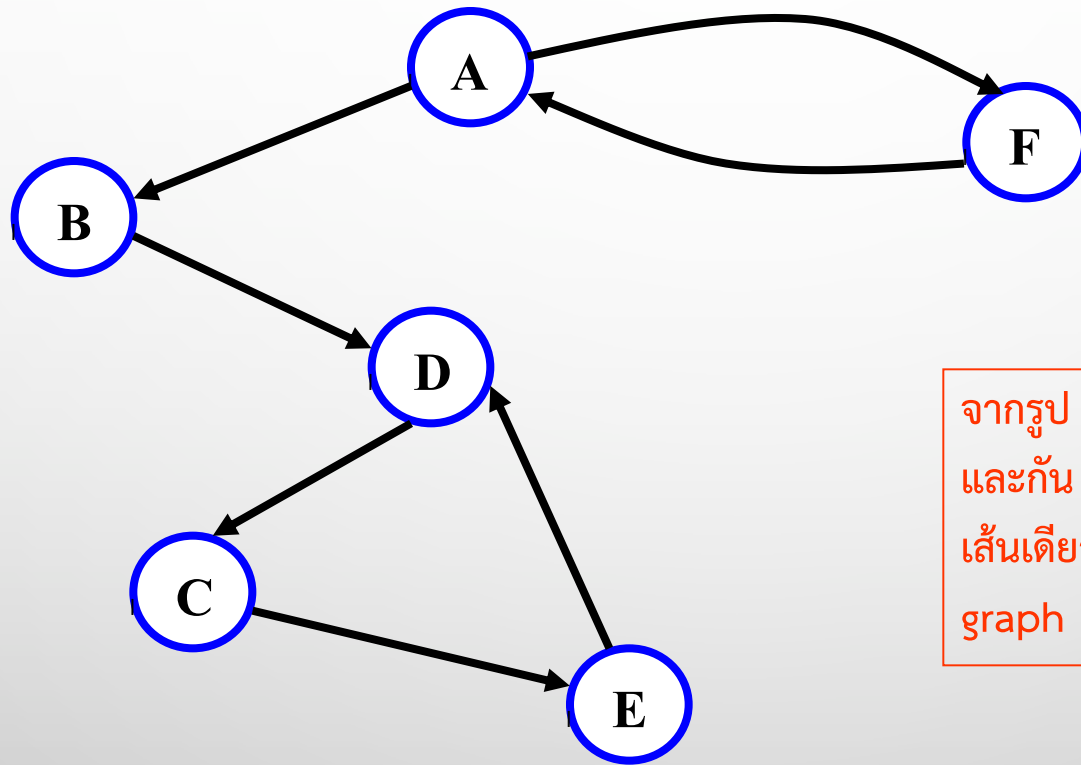
*Directed graph หรือ digraph* เป็นกราฟที่เส้นเชื่อม (edge) ระหว่างคู่ของโนดมีลูกศร (arrow) แสดงทิศทางจากโนดแรกในคู่ ไปยังโนดที่สอง (ถูกชี้ด้วยลูกศร) และ เราเรียก edge ของ digraph ว่า *arcs*

# กราฟไม่มีทิศทาง (An Undirected Graph)



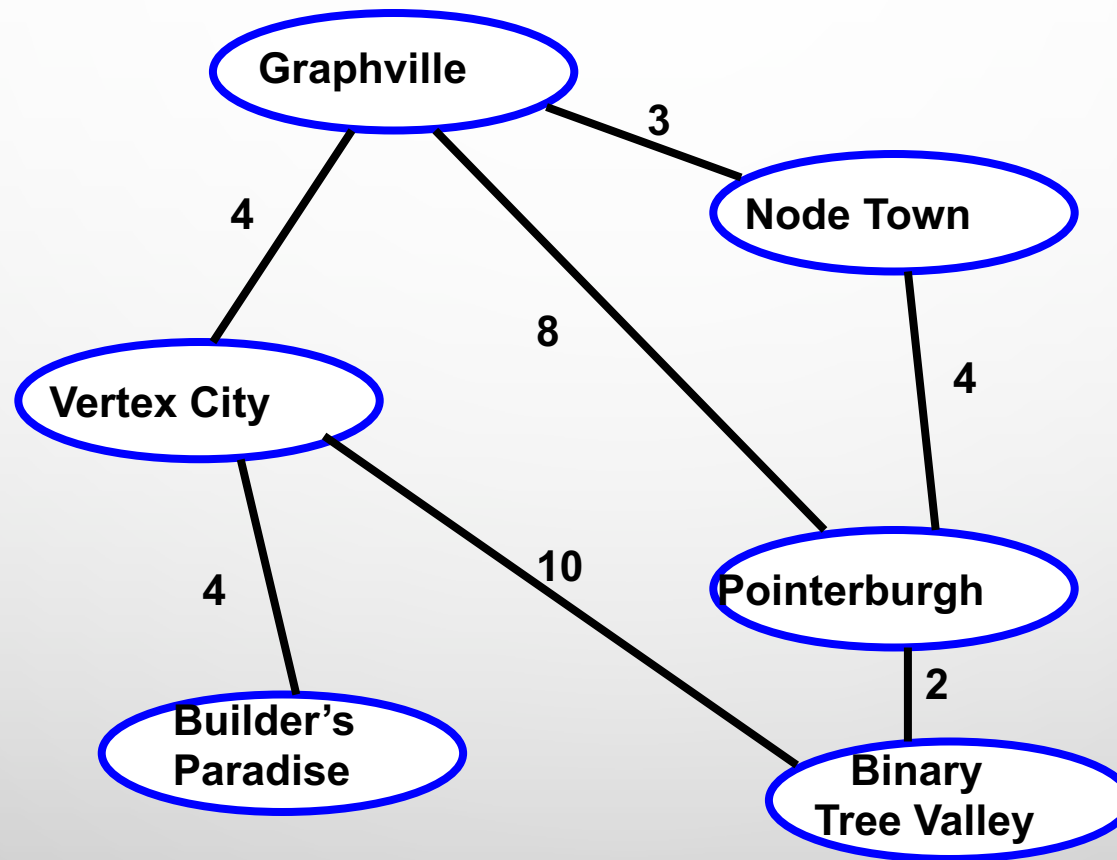
**Undirected graph** เป็นกราฟที่ไม่มีทิศทางระหว่าง 2 vertices โดยสามารถไปได้ทั้ง 2 ทิศทาง จากภาพ แต่ละ edge จะชี้ไปสองทิศทาง กล่าวคือ A ชี้ไปที่ D และ D ชี้ไปที่ A

# รูปแบบอื่นของ Digraph



จากรูป โหนด A และ F ชี้ไปซึ่งกัน  
และกัน – เราจะไม่วาดเส้นตรงเพียง  
เส้นเดียวเหมือนกับ undirected  
graph

# กราฟแบบมีน้ำหนัก (Weighted Graph)



# PATH

- **PATH** หมายถึง เส้นทางที่มีการเชื่อมต่อกันของโนด หรือ ลำดับของ EDGE ที่เชื่อมต่อไปในทิศทางเดียวกัน
- PATH จากโหนดหนึ่งไปยังตัวมันเองเรียกว่า **CYCLE**.
- ถ้ากราฟมี CYCLE จะเป็น **CYCLIC GRAPH** ถ้าไม่มีเป็น **ACYCLIC GRAPH**

## จากรูปมี

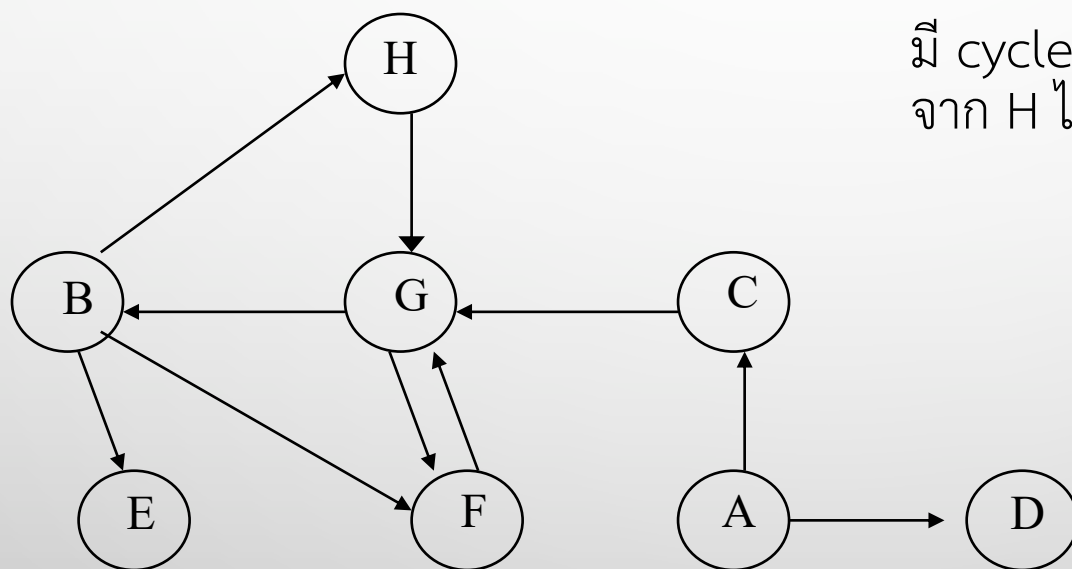
1 path ความยาว 1 จาก A ไป C

2 paths ความยาว 2 จาก B ไป G

1 path ความยาว 3 จาก A ไป F

ไม่มี path จาก B ไป C

มี cycle จาก B ไป B จาก F ไป F และ  
จาก H ไป H



# การนำกราฟไปใช้แก้ปัญหา

**โจทย์** : หาเส้นทางบินตรงจากเมืองหนึ่งไปเมืองหนึ่งโดยผ่านเมืองอื่นน้อยที่สุด

**วิธีการแก้ปัญหา** : ใช้ DIRECTED GRAPH ในการโมเดลเครือข่ายของเส้นทางการบินของสายการบิน โดย

- ให้ VERTICES แทนชื่อเมืองที่มีสนามบิน
- DIRECT ARCS แสดงเส้นทางบินที่เชื่อมระหว่างเมือง
- การหาเส้นทางบินตรงเท่ากับการหา SHORTEST PATH หรือหาจำนวน ARCS ที่น้อยที่สุดจาก VERTEX เริ่มต้นไปยัง VERTEX ปลายทาง



# โครงสร้างที่ใช้เก็บ Graph

- **Adjacent vertices** คือ 2 vertices ใดๆในกราฟที่ถูกเชื่อมต่อด้วย edge เรากล่าวว่า vertex A *adjacent* กับ vertex B เมื่อมี edge ชี้จาก A ไป B
- โครงสร้างที่นิยมใช้ในการสร้างกราฟได้แก่
  - adjacency matrix
  - adjacency list

# Adjacency Matrix

- Adjacency matrix

เป็นอะเรย์สองมิติของตัวแปรที่มีค่าเป็น Boolean

- โหนดในกราฟจะถูกให้ค่าตั้งแต่ 0 ถึง  $n-1$  โดยที่  $n$  เป็นจำนวนโนดทั้งหมดในกราฟ

	0	1	2	3	4	5
0	T	T	F	F	F	F
1	F	T	T	F	F	T
2	F	F	T	T	T	F
3	F	F	F	T	F	F
4	F	T	F	T	T	T
5	F	F	F	F	F	T

# Adjacency Matrix (cont.)

	0	1	2	3	4	5
0	T	T	F	F	F	F
1	F	T	T	F	F	T
2	F	F	T	T	T	F
3	F	F	F	T	F	F
4	F	T	F	T	T	T
5	F	F	F	F	F	T

- **แถว** แสดงหมายเลขของ vertex ที่มี edge ซึ่ออก
- **คอลัมน์** แสดงหมายเลขของ vertex ที่มี edge ซึ่เข้า
- ค่าในจุดตัดของแถวและคอลัมน์
  - ถ้าเป็น **T** แสดงว่ามี edge ระหว่างโนดหมายเลข (แถว) ซึ่ไปยังโนดหมายเลข (คอลัมน์)
  - ถ้าเป็น **F** แสดงว่าไม่มี edge ระหว่างโนดหมายเลข (แถว) ซึ่ไปยังโนดหมายเลข (คอลัมน์)

# Adjacency Matrix (cont.)

	0	1	2	3	4	5
0	T	T	F	F	F	F
1	F	T	T	F	F	T
2	F	F	T	T	T	F
3	F	F	F	T	F	F
4	F	T	F	T	T	T
5	F	F	F	F	F	T

ตัวอย่าง :

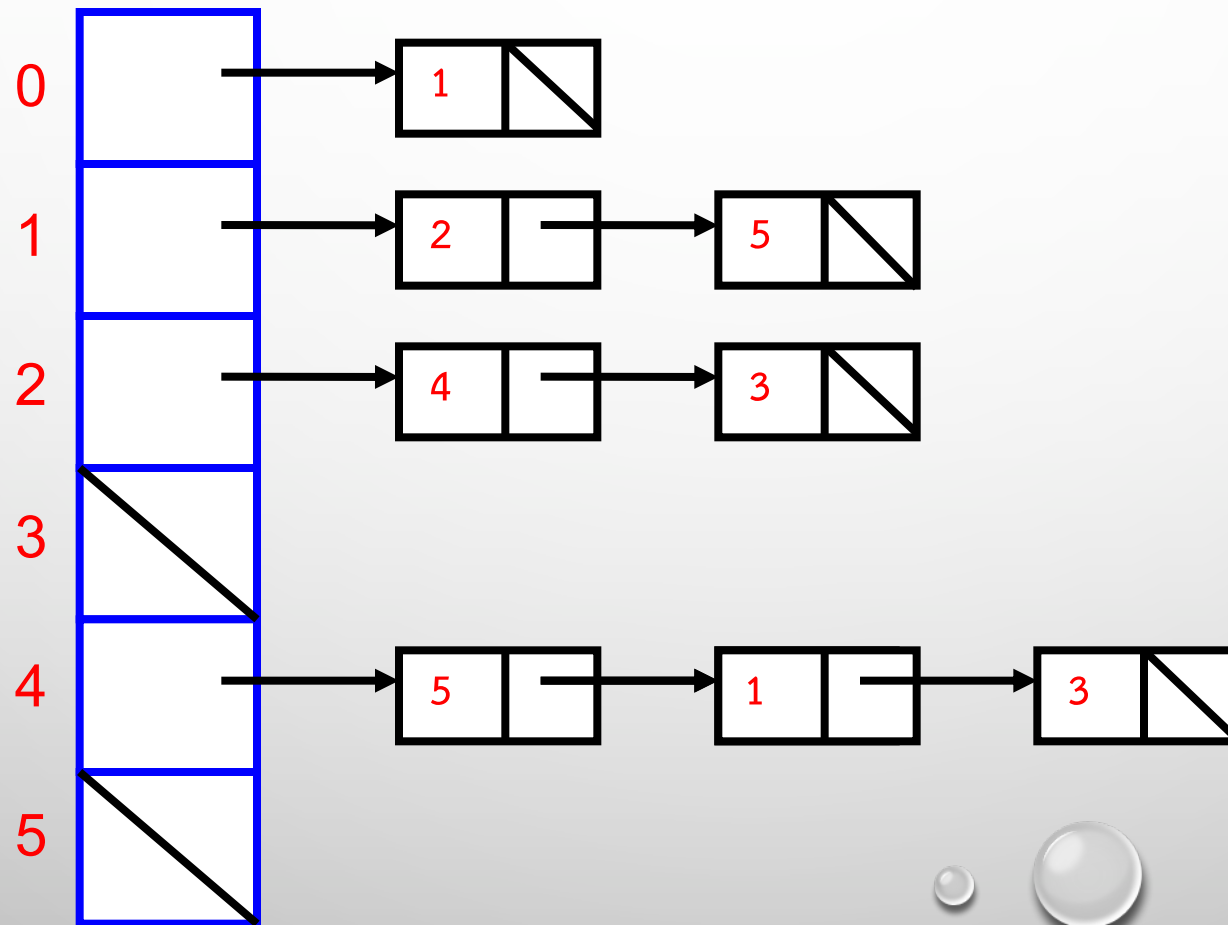
โนด 1 ซึ่ไปยังโนด 2 (ค่าเป็น T) แต่  
โนด 2 ไม่ได้ซึ่ไปยังโนด 1

หมายเหตุ : เราสามารถวาดกราฟได้  
จาก adjacency matrix โดยตำแหน่ง  
ของโนดอาจไม่ได้เรียงลำดับเหมือนเดิม  
แต่เส้นทางเชื่อมของโนดต้องเหมือนกัน

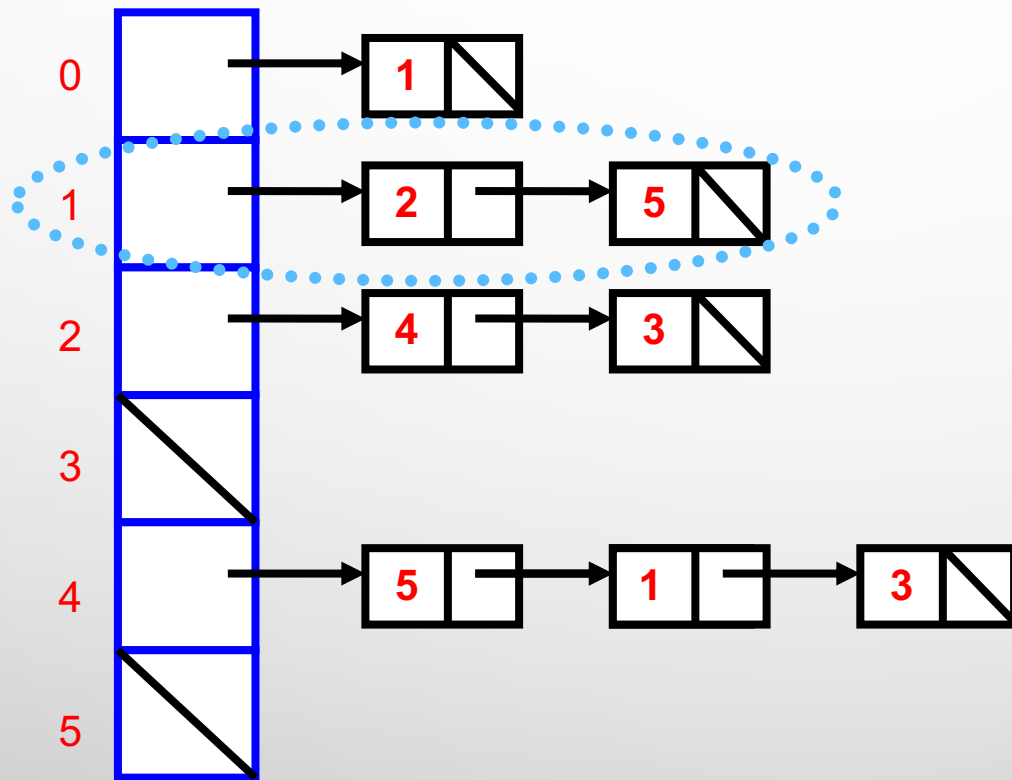
# Adjacency List

- Adjacency list เป็นอะเรย์ของ linked lists
- Vertices จะถูกให้หมายเลขตั้งแต่ 0 ถึง  $n - 1$
- หมายเลข index ในอะเรย์จะเป็นที่เก็บข้อมูลของ vertex หมายเลขเดียวกัน
- Vertex ที่อะเรย์ช่องใดๆ จะเชื่อมต่อกับโนดที่อยู่ใน linked list ของช่องนั้น

# Adjacency List (cont.)



# Adjacency List (cont.)

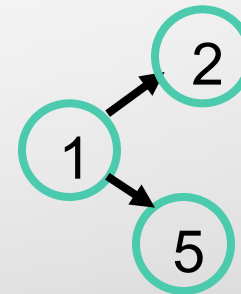


จากรูป:

Vertex 1 is **adjacent** to vertex 2

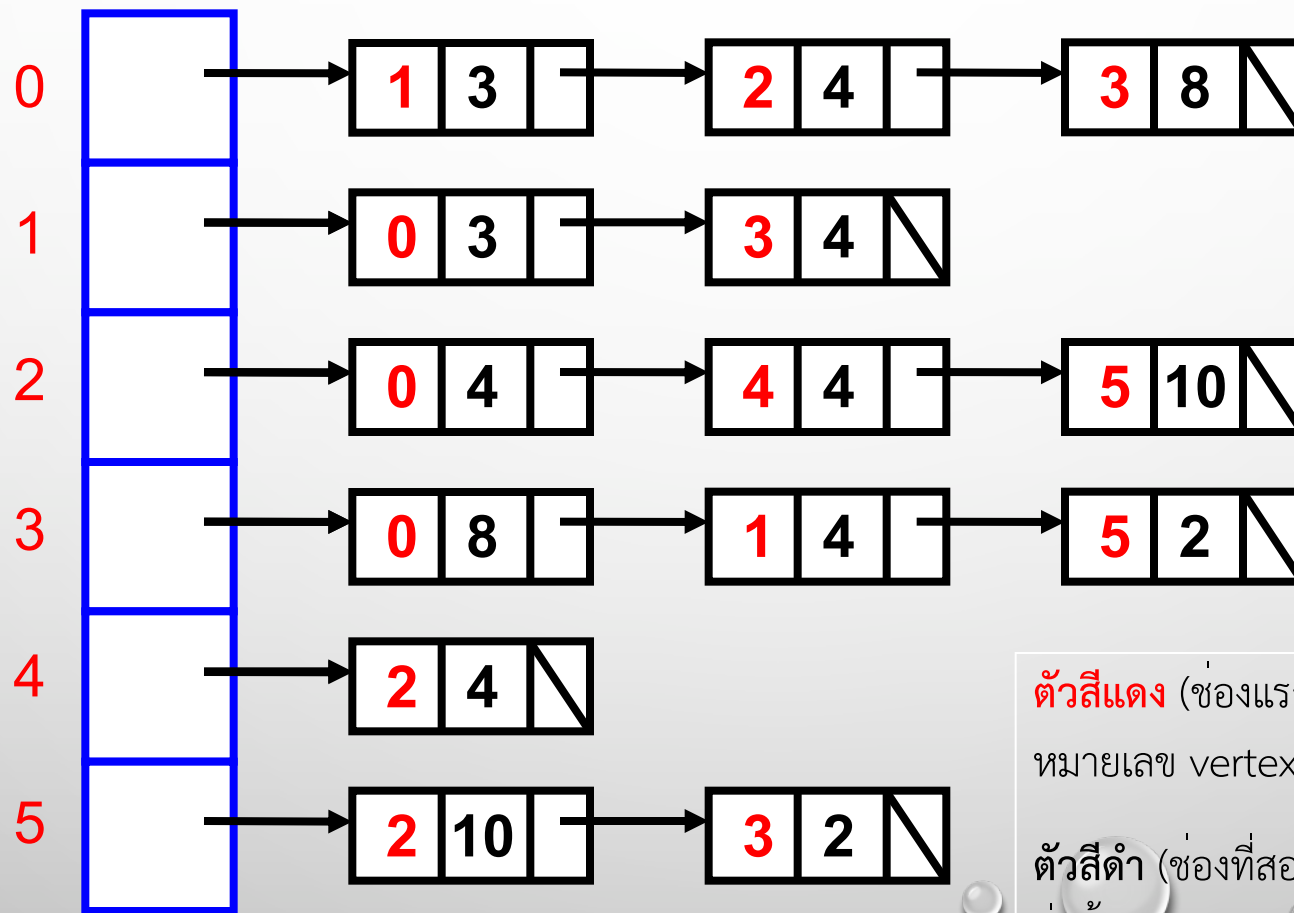
Vertex 1 is **adjacent** to vertex 5

Vertex 2 อาจจะเชื่อมหรือไม่เชื่อมต่อกับ Vertex 5 ก็ได้ ดังภาพด้านล่างนี้



Vertex 3 ลิสต์เป็นลิสต์ว่างหมายถึงไม่เชื่อมต่อกับ vertex ใดเลย

# Adjacency List for Weighted Graph



ตัวสีแดง (ช่องแรกของโนด) หมายถึง  
หมายเลข vertex

ตัวสีดำ (ช่องที่สองของโนด) หมายถึง  
ค่าน้ำหนัก(weights)



# การเก็บข้อมูลของ Vertex

- โดยปกติแล้ว เราจะไม่เก็บข้อมูลอื่นๆ นอกจากหมายเลขของ vertex ไว้ในโนดของลิสต์ใน adjacency list เนื่องจากข้อมูลอื่นของ vertex อาจจะใช้เนื้อที่ขนาดใหญ่ และจะเกิดความซ้ำซ้อนของข้อมูลใน adjacency list เพราะในหลายลิสต์อาจจะมีเก็บข้อมูลของ vertex เดียวกันทำให้เกิดความซ้ำซ้อนและเปลืองเนื้อที่
- ดังนั้น ถ้าต้องการเก็บข้อมูลอื่นของ vertex เรามักจะแยกเก็บในอะเรย์ต่างหาก โดยใช้หมายเลข index ของอะเรย์นี้เป็นตัวเชื่อมโยงไปยัง vertex ที่เก็บอยู่ในอะเรย์ของ adjacency list

# Adjacency Matrix vs. Adjacency List

- ความเร็วของการใช้ adjacency matrix หรือ adjacency list ขึ้นกับอัลกอริทึมที่ใช้
  - ถ้าอัลกอริทึมที่ใช้ต้องการรู้ว่ามีเส้นทางเชื่อมตรงระหว่างสอง vertices หรือไม่ การใช้ adjacency matrix จะทำงานได้เร็วกว่า
  - ถ้าอัลกอริทึมที่ใช้ถูกเขียนมาเพื่อประมวลผลลิงค์ลิสต์ใน adjacency list แบบที่ละโนด การใช้ adjacency list จะทำงานได้เร็วกว่า
- ถ้าความเร็วของการใช้ adjacency matrix หรือ adjacency list ใกล้เคียงกัน ให้พิจารณาจากพื้นที่หน่วยความจำที่ต้องใช้ (space complexity)
  - **Space complexity** จะเป็นการพิจารณาถึงเนื้อที่ของหน่วยความจำที่ต้องใช้เมื่อขนาดของปัญหา (ขนาดของข้อมูล) เปลี่ยนไป

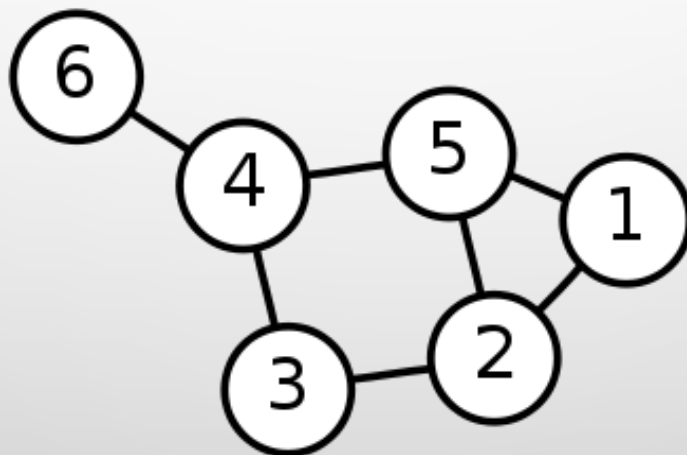
The background of the slide is a light gray gradient. It is decorated with numerous realistic water droplets of various sizes. Some droplets are at the top left, some are scattered in the middle, and a larger cluster of droplets is in the bottom right corner. The droplets have highlights and shadows, giving them a three-dimensional appearance.

# DIJKSTRA'S ALGORITHM

SLIDE COURTESY: UWASH, UT

# DIJKSTRA'S ALGORITHM

ใช้ในการแก้ปัญหา **SINGLE-SOURCE SHORTEST PATH** หรือ  
การหาเส้นทางที่สั้นที่สุดจากจุดเริ่มต้นเพียงจุดเดียวไปยังจุดอื่นๆ  
ทั้งหมดในกราฟ



# DIJKSTRA'S ALGORITHM

- สามารถทำงานได้กับทั้ง DIRECTED AND UNDIRECTED GRAPHS
- แต่ทุก EDGES ต้องมีน้ำหนักไม่เป็นค่าลบ

**INPUT:** กราฟแบบมีน้ำหนัก  $G=\{E,V\}$  และ จุดเริ่มต้น  $v \in V$  โดยที่ค่าน้ำหนักของทุก EDGE ไม่เป็น  
เลขลบ

**OUTPUT:** ความยาวของ PATH ที่สั้นที่สุดจากจุดเริ่มต้น  $v \in V$  ไปยังจุดอื่นๆ ทั้งหมดในกราฟ

# DIJKSTRA PSEUDOCODE

*DIJKSTRA(V1, V2):*

*FOR EACH VERTEX V:     // INITIALIZATION*

*V'S DISTANCE := INFINITY.*

*V'S PREVIOUS := NONE.*

*V1'S DISTANCE := 0.*

*LIST := {ALL VERTICES}.*

*WHILE LIST IS NOT EMPTY:*

*V := REMOVE LIST VERTEX WITH MINIMUM DISTANCE.*

*MARK V AS KNOWN.*

*FOR EACH UNKNOWN NEIGHBOR N OF V:*

*DIST := V'S DISTANCE + EDGE (V, N)'S WEIGHT.*

*IF DIST IS SMALLER THAN N'S DISTANCE:*

*N'S DISTANCE := DIST.*

*N'S PREVIOUS := V.*

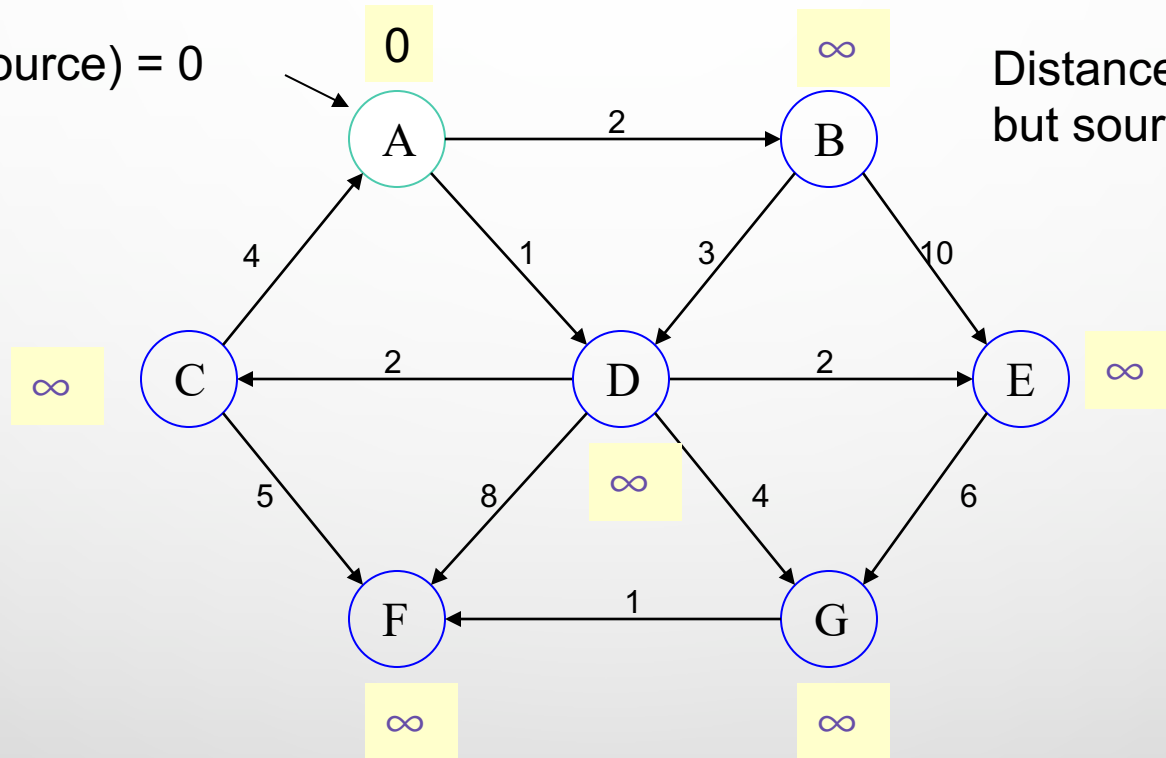
*RECONSTRUCT PATH FROM V2 BACK TO V1,*

*FOLLOWING PREVIOUS POINTERS.*

# EXAMPLE: INITIALIZATION

V	Dist(V)	N_Prev
A	0	
B	$\infty$	
C	$\infty$	
D	$\infty$	
E	$\infty$	
F	$\infty$	
G	$\infty$	

Distance(source) = 0



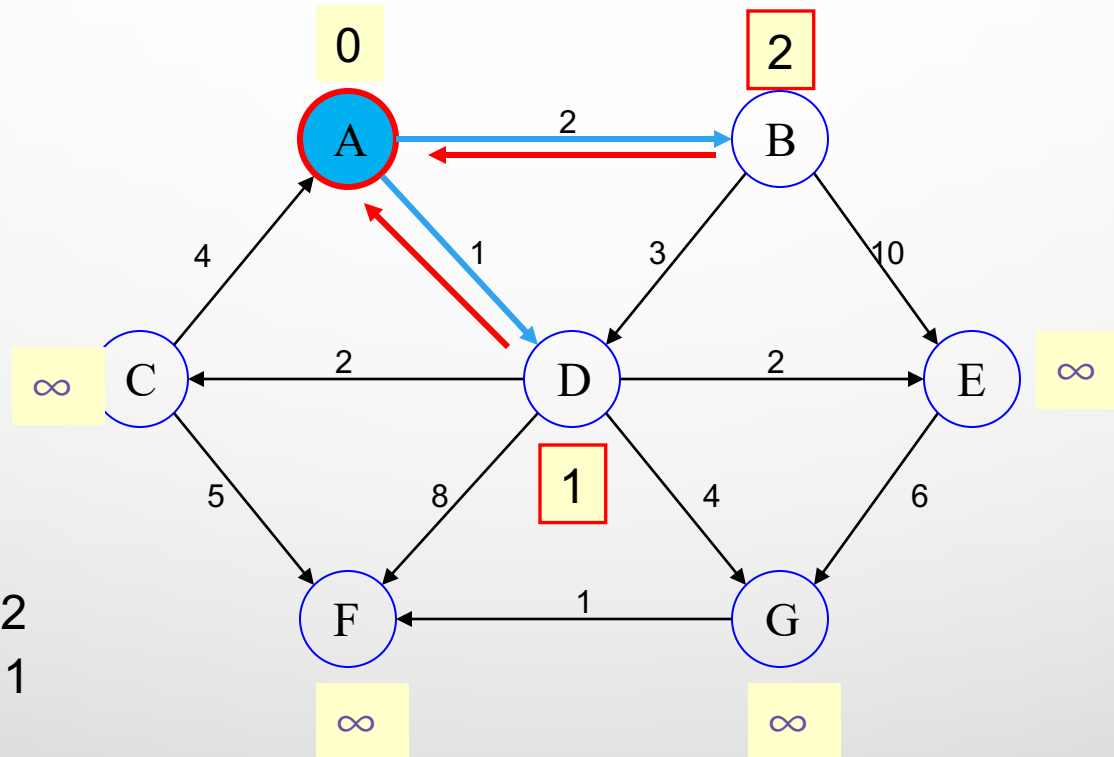
ให้ระยะทางที่จุดเริ่มต้นเป็น 0 ส่วนจุดอื่นๆ เป็นค่า  $\infty$

## EXAMPLE: UPDATE NEIGHBORS' DISTANCE

V	Dist(V)	N_Prev
A	0	
B	$\infty$ 2	A
C	$\infty$	
D	$\infty$ 1	A
E	$\infty$	
F	$\infty$	
G	$\infty$	

Distance(B) = 2

Distance(D) = 1

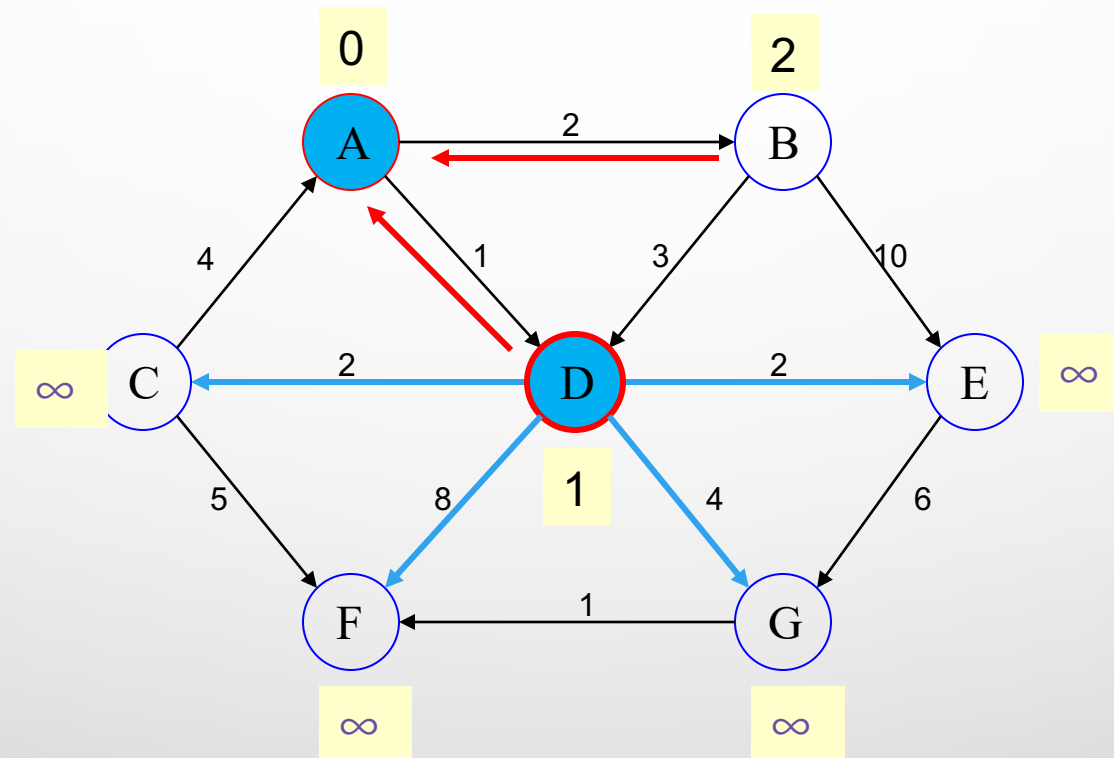


อัปเดตระยะทางของจุดที่จุด A สามารถไปถึง ได้แก่ B และ D



## EXAMPLE: REMOVE VERTEX WITH MINIMUM DISTANCE

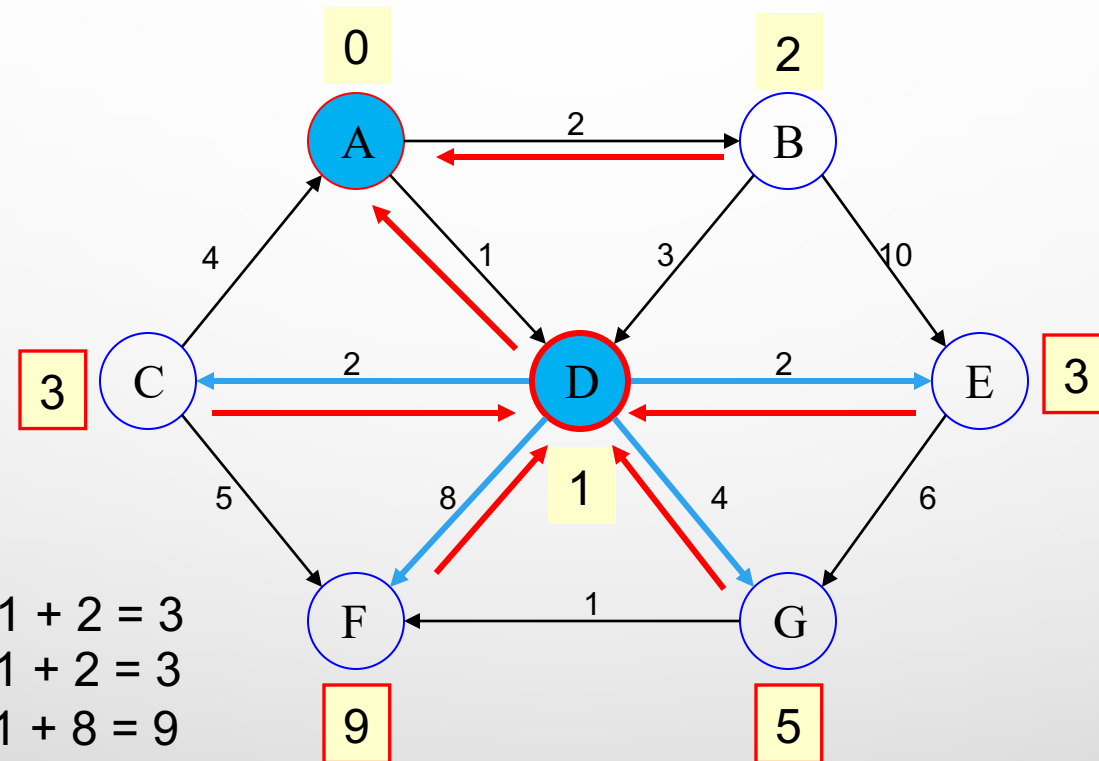
V	Dist(V)	N_Prev
A	0	
B	$\infty$ 2	A
C	$\infty$	
D	$\infty$ 1	A
E	$\infty$	
F	$\infty$	
G	$\infty$	



เลือกจุดที่ให้ค่าระยะทางสั้นที่สุดในการดำเนินการต่อ ได้แก่ จุด D

# EXAMPLE: UPDATE NEIGHBORS

V	Dist(V)	N_Prev
A	0	
B	$\infty$ 2	A
C	$\infty$ 3	D
D	$\infty$ 1	A
E	$\infty$ 3	D
F	$\infty$ 9	D
G	$\infty$ 5	D



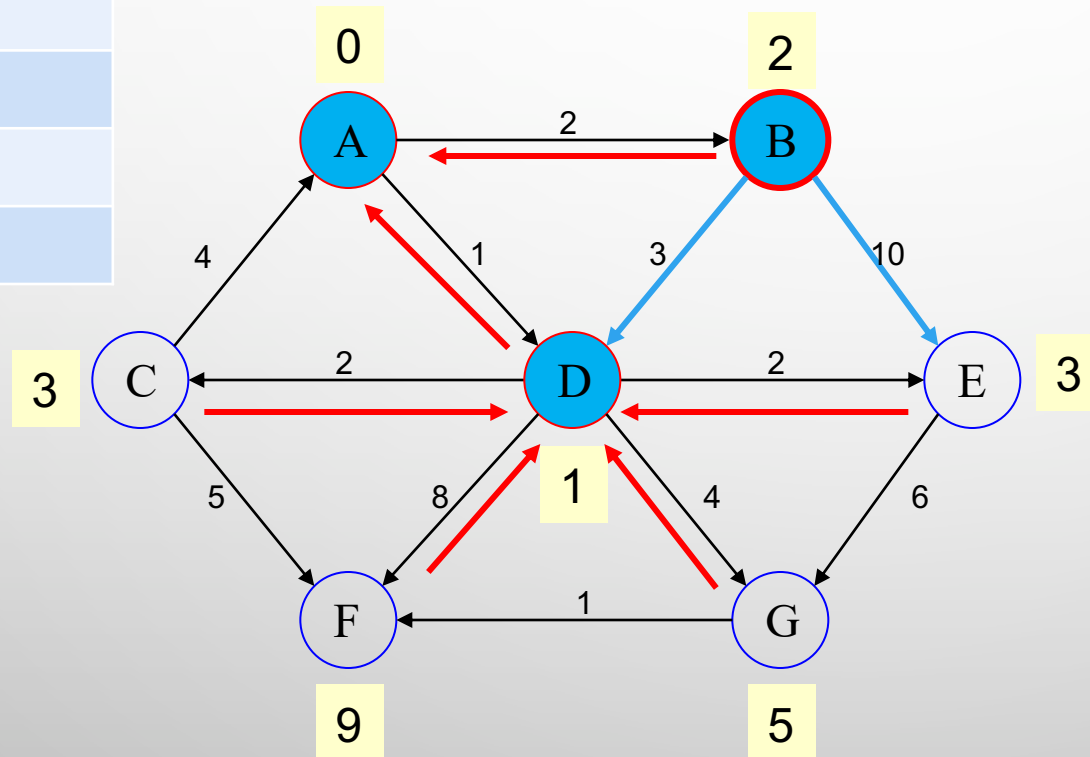
Distance(C) = 1 + 2 = 3  
Distance(E) = 1 + 2 = 3  
Distance(F) = 1 + 8 = 9  
Distance(G) = 1 + 4 = 5

อัปเดตระยะทางของจุดที่จุด D สามารถไปถึง ได้แก่ C, E, F และ G

# EXAMPLE: CONTINUED...

V	Dist(V)	N_Prev
A	0	
B	$\infty$ 2	A
C	$\infty$ 3	D
D	$\infty$ 1	A
E	$\infty$ 3	D
F	$\infty$ 9	D
G	$\infty$ 5	D

เลือกจุดที่ให้ค่าระยะทางสั้นที่สุดในการดำเนินการต่อ ได้แก่ จุด B  
และอัปเดตค่าระยะทางของจุดที่ B ไปถึง ได้แก่ D และ E

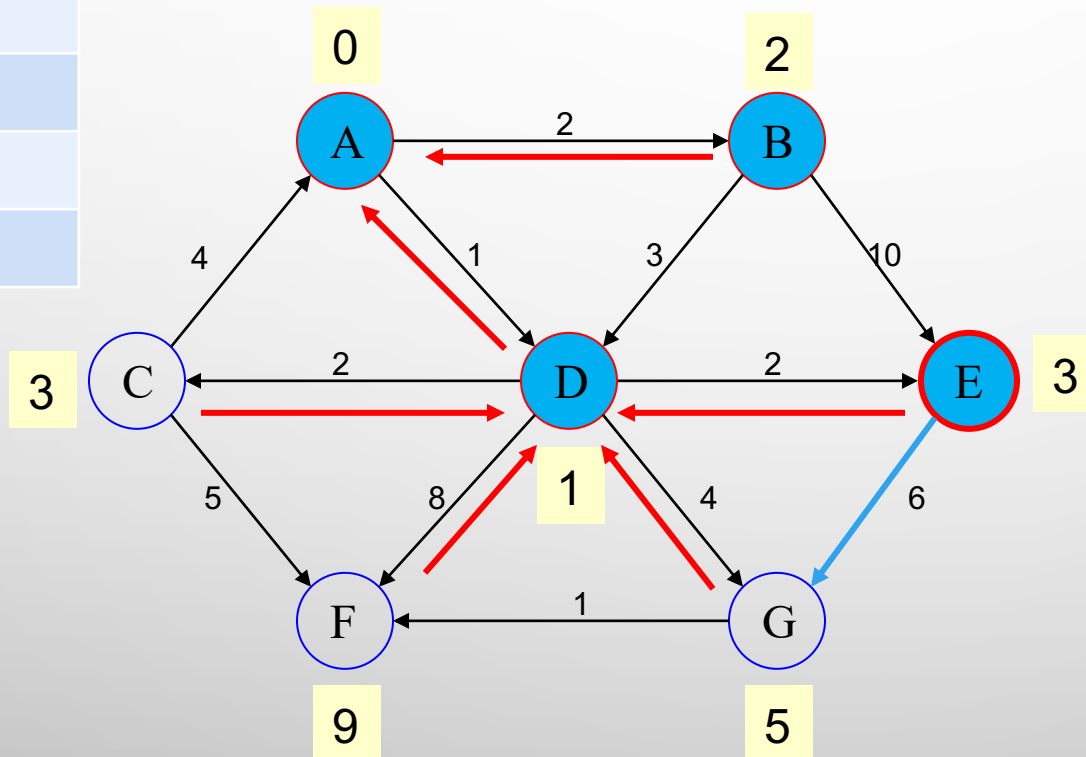


หมายเหตุ: distance(D) not  
ไม่ต้องอัปเดตเนื่องจาก D เป็น  
จุดที่ผ่านไปแล้ว ส่วน  
distance(E) ก็ไม่ต้องอัปเดต  
เช่นกัน เนื่องจากมีค่าที่คำนวณ  
ได้มีค่ามากกว่าค่าเดิม

## EXAMPLE: CONTINUED...

V	Dist(V)	N_Prev
A	0	
B	$\infty$ 2	A
C	$\infty$ 3	D
D	$\infty$ 1	A
E	$\infty$ 3	D
F	$\infty$ 9	D
G	$\infty$ 5	D

เลือกจุดที่ให้ค่าระยะทางสั้นที่สุดในการดำเนินการต่อ ได้แก่ จุด E  
และอัปเดตค่าระยะทางของจุดที่ E ไปถึง ได้แก่ G

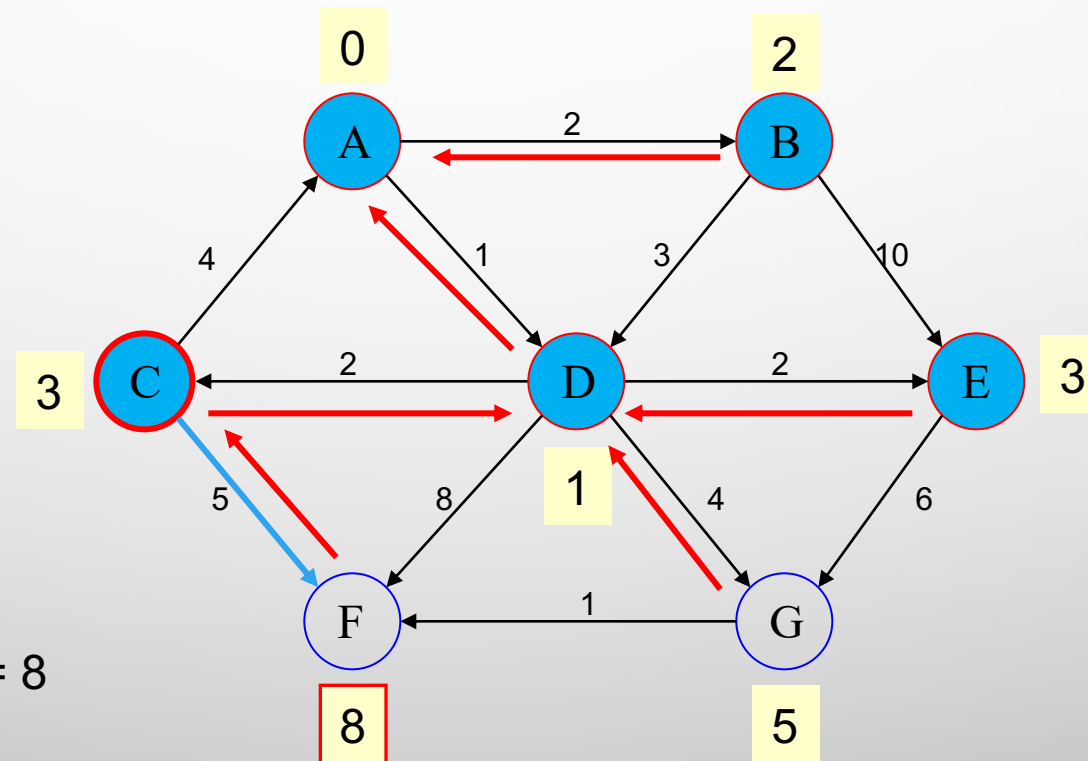


หมายเหตุ: ไม่มีการ  
อัปเดต distance(G)  
เนื่องจากค่าที่คำนวณได้  
มากกว่าค่าเดิม

## EXAMPLE: CONTINUED...

เลือกจุดที่ให้ค่าระยะทางสั้นที่สุดในการดำเนินการต่อ ได้แก่ จุด C  
และอัปเดตค่าระยะทางของจุดที่ C ไปถึง ได้แก่ F

V	Dist(V)	N_Prev
A	0	
B	∞ 2	A
C	∞ 3	D
D	∞ 1	A
E	∞ 3	D
F	∞ 9 8	D C
G	∞ 5	D

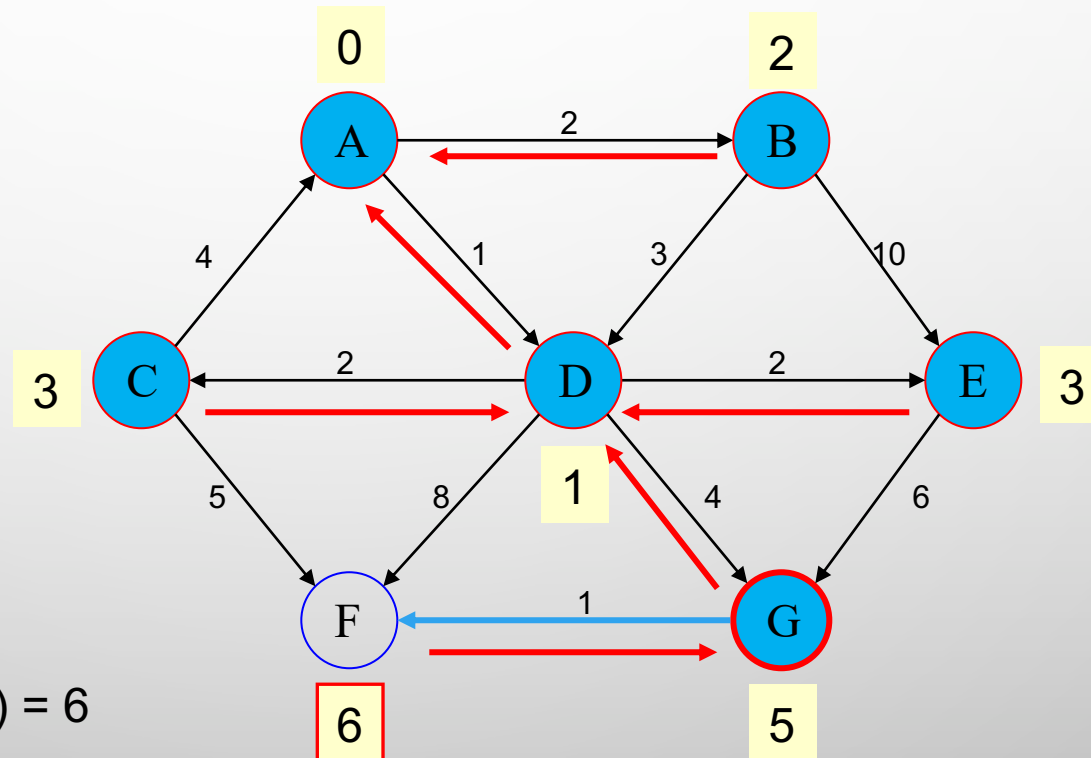


$$\text{Distance}(F) = 3 + 5 = 8$$

## EXAMPLE: CONTINUED...

เลือกจุดที่ให้ค่าระยะทางสั้นที่สุดในการดำเนินการต่อ ได้แก่ จุด G  
และอัปเดตค่าระยะทางของจุดที่ G ไปถึง ได้แก่ F

V	Dist(V)	N_Prev
A	0	
B	∞ 2	A
C	∞ 3	D
D	∞ 1	A
E	∞ 3	D
F	∞ 9 8 6	D ∈ G
G	∞ 5	D

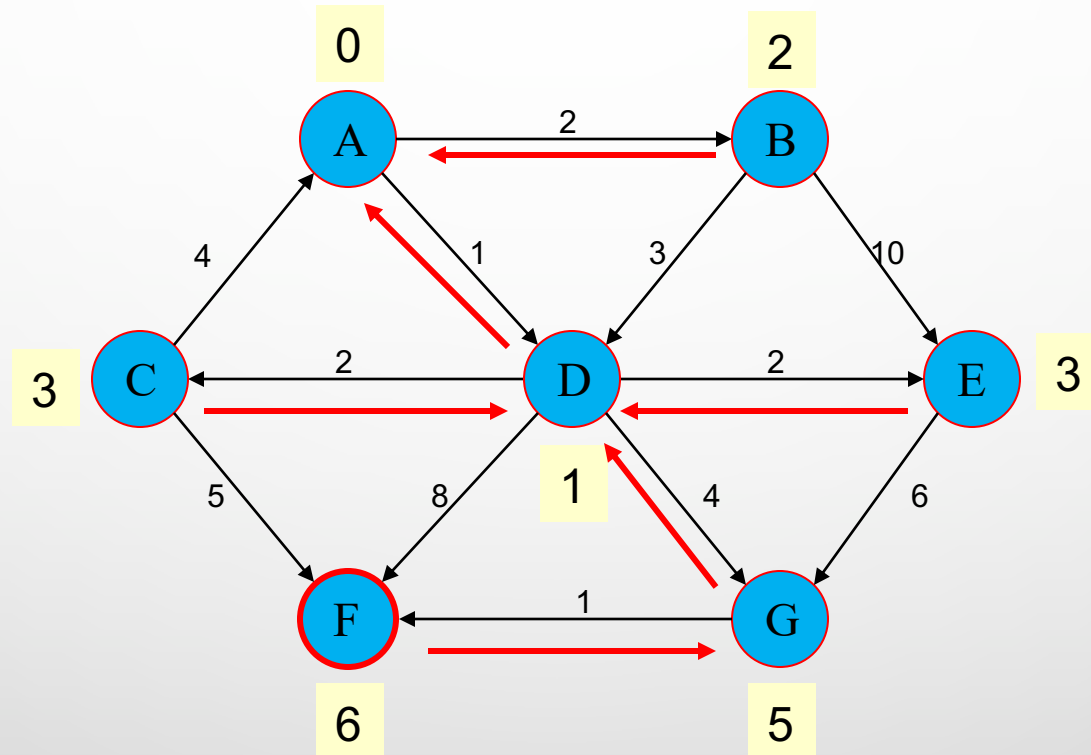


Previous distance

$$\text{Distance}(F) = \min(8, 5+1) = 6$$

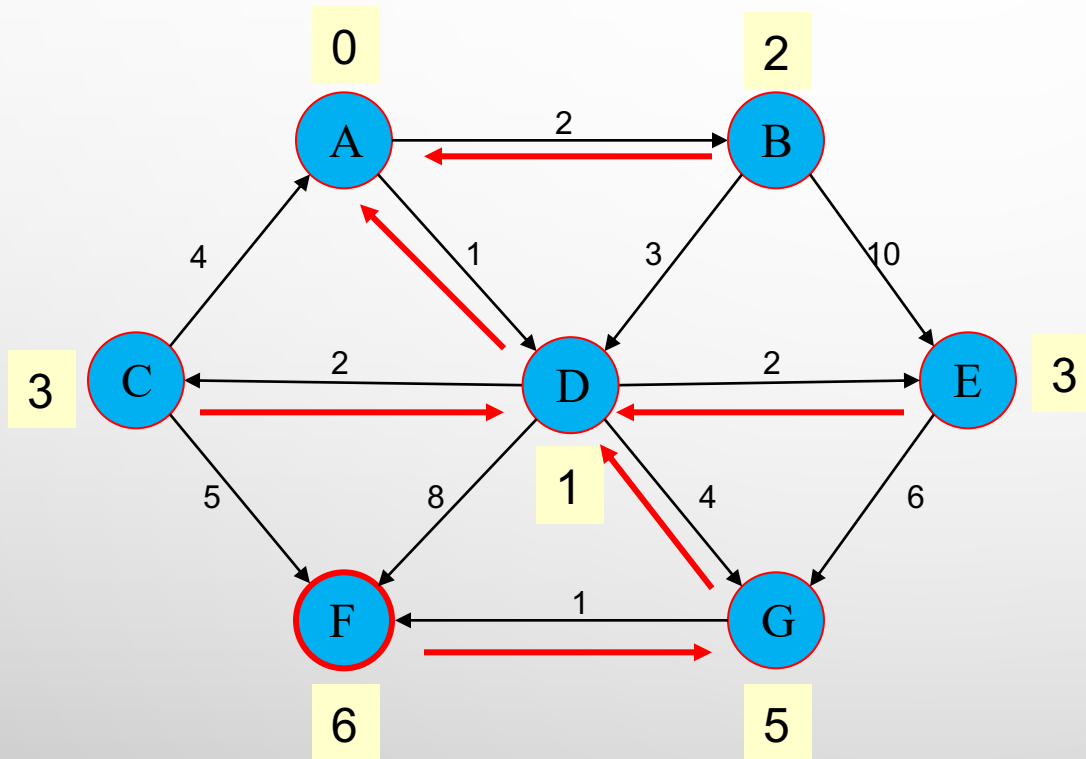
# EXAMPLE (END)

V	Dist(V)	N_Prev
A	0	
B	$\infty$ 2	A
C	$\infty$ 3	D
D	$\infty$ 1	A
E	$\infty$ 3	D
F	$\infty$ 9 8 6	$\emptyset \in G$
G	$\infty$ 5	D



เลือกจุดที่เหลือ ได้แก่ F และอัปเดตจุดที่ F ไปถึง (ไม่มี)

# EXAMPLE : CONCLUSION



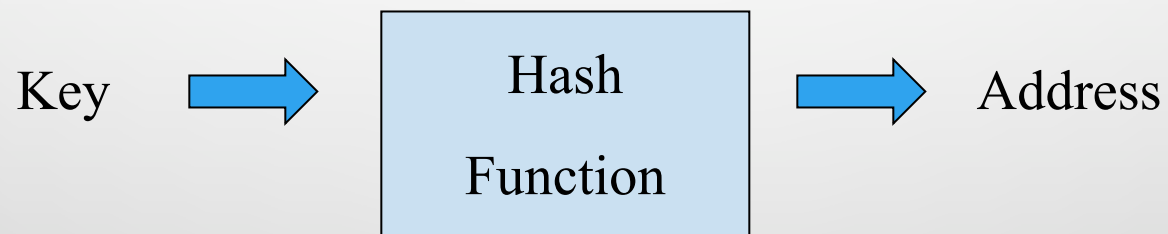
เส้นทาง	ระยะที่สั้นที่สุด
A->B	2
A->C	3
A->D	1
A->E	3
A->F	6
A->G	5



# Hashing

# แนวคิด

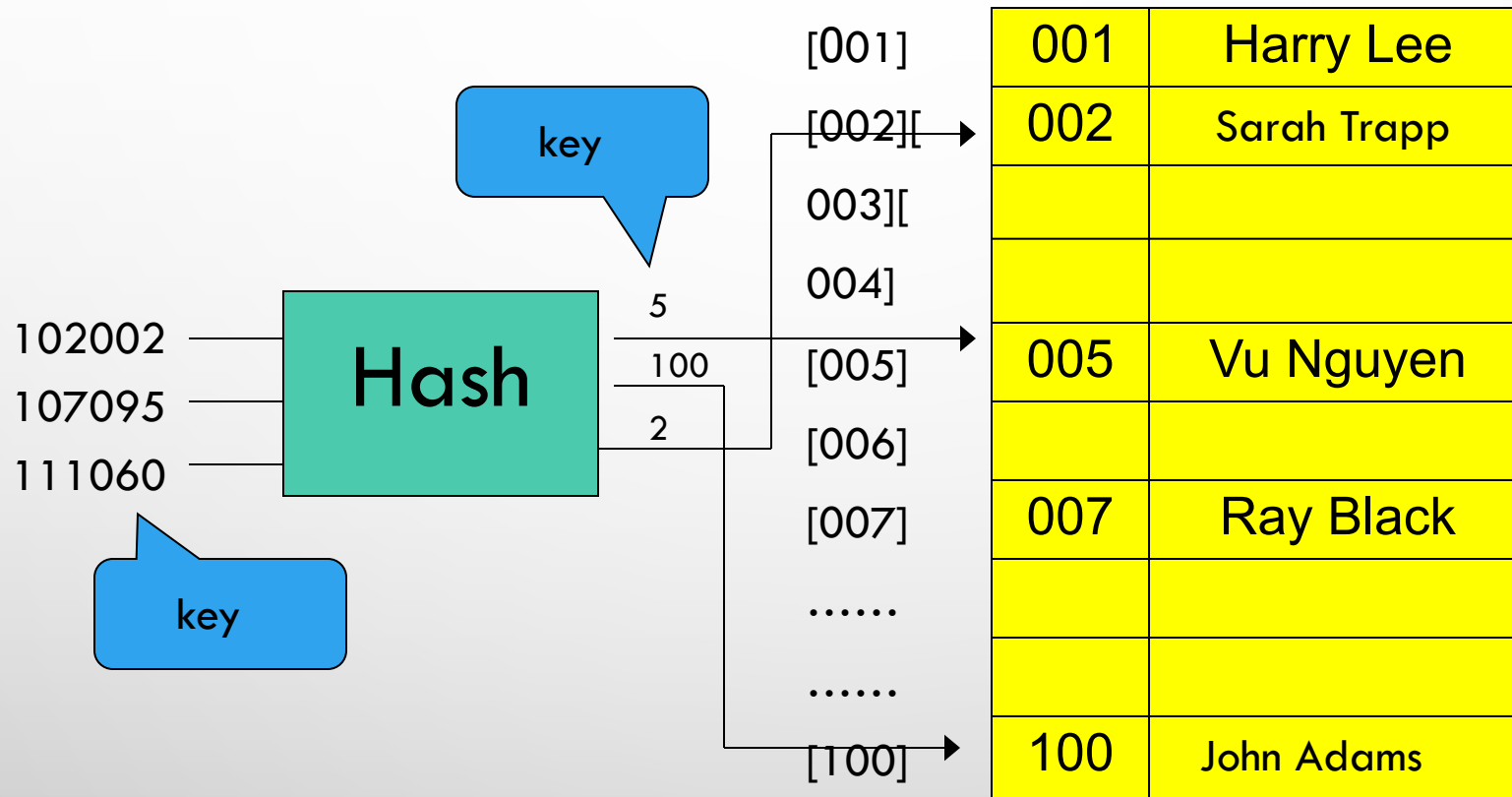
- วิธีการค้นหาข้อมูลส่วนใหญ่ต้องการการเปรียบเทียบข้อมูลหลายครั้งกว่าจะได้พบข้อมูลที่ต้องการ
- ถ้าเป็นไปได้เราต้องการรู้ว่าข้อมูลที่ต้องการเก็บอยู่ที่ใด แล้วไปที่ตำแหน่งนั้นโดยตรง
- วิธีการข้างต้น เรียกว่า HASH SEARCH ซึ่งจะมีอัลกอริทึมเพื่อกำหนดตำแหน่งที่เก็บข้อมูล



# นิยาม

- Hash table เป็นโครงสร้างอะเรย์ที่เก็บข้อมูล  
โดยทั่วไปจะใช้ *key* (บางส่วนของข้อมูล) เพื่อกำหนดตำแหน่งที่จะ  
จัดเก็บในตาราง
- Hashing เป็นเทคนิคที่ใช้ในการแทรก ลบและค้นหาข้อมูลด้วยเวลาเฉลี่ยคงที่
- Hash function เป็นฟังก์ชันการแปลง *key* ให้เป็นจำนวนที่มีค่าอยู่ระหว่าง 0  
ถึง  $\text{TableSize} - 1$  และจัดวางข้อมูลในตำแหน่งที่ได้
- *TableSize* หมายถึงขนาดของ hash table.

# ตัวอย่าง HASH TABLE



- เนื่องจากจำนวนช่องที่จะเก็บข้อมูลมีจำนวนจำกัด ดังนั้น hash function ที่ดีควรจะสามารถกระจาย keys ไปได้ทั่วตาราง
- ปัญหาชวนคิด:
  - เราจะสามารถแปลง key ที่อาจเป็นข้อความให้เป็นตำแหน่งซึ่งเป็นตัวเลขได้อย่างไร
  - เราจะสามารถแก้ปัญหาคollision ได้อย่างไร
  - ขนาดของ table size ควรเป็นเท่าใด

# เทคนิคการทำ HASHING

สามารถแบ่งได้เป็น 2 กลุ่ม

- **Simple**
  - Direct (and subtraction)
  - Modulo Division
  - Random Number Generation
- **Permuting** (มักใช้ร่วมกับวิธี modulo division)
  - Digit Extraction
  - Midsquare
  - Folding
  - Rotation

# DIRECT AND SUBTRACTION HASHING

- สมมติว่าคีย์ทั้งหมดเป็นตัวเลขและมีลำดับ
- ใช้ keys สำหรับบอกตำแหน่งโดยตรง หรือลบ (subtract) ด้วยเลขบางตัว เพื่อให้ได้ตำแหน่งใน hash table.
- วิธีนี้อาจไม่สามารถใช้ได้ ในบางสถานการณ์ แต่ในบางสถานการณ์สามารถใช้ได้ดี ถ้าเราแน่ใจว่าจะไม่เกิด synonyms

# DIRECT AND SUBTRACTION HASHING

## ตัวอย่าง

- สมมติว่าต้องการเก็บยอดขายสินค้าในแต่ละวันของเดือน เราต้องการอะเรย์เพื่อเก็บข้อมูลขนาด 31 ช่อง เราสามารถใช้วันที่ของยอดขายนั้นเป็น key เช่น ยอดขายของวันที่ 1 เก็บในช่องที่ 1 เป็นต้น
- องค์กรที่มีจำนวนพนักงานไม่เกิน 100 คน และเลขที่พนักงานอยู่ระหว่าง 1-100 สามารถใช้อะเรย์ขนาด 100 ช่องในการเก็บข้อมูลโดยใช้เลขที่พนักงานเป็นตัวบอกตำแหน่ง
- หมายเลขเช็คมักจะเริ่มต้นด้วย 101 หมายเลขเช็คสามารถถูก hash ด้วยการลบด้วย 100 เพื่อให้ได้ตำแหน่ง



# MODULO DIVISION

- มักใช้ในกรณีที่ค่าของ key มากกว่าขนาดของ hash table เช่น เราต้องการ hash ข้อมูลที่เป็นตัวเลข 4 หลักให้สามารถเก็บลงในตารางข้อมูลที่มีเพียง 100 ช่อง

ตำแหน่ง =  $\text{key} \text{ MODULUS } \text{ขนาดของ hash table}$

เช่น  $1234 / 100 = 12$  ได้เศษ 34 ตำแหน่งที่ใช้เก็บ คือ 34

- โดยปกติแล้ว ถ้าให้ขนาดของ hash table เป็น prime number โอกาสที่เกิด collision มีน้อยลง จากตัวอย่างข้างต้นขนาดควรเป็น 101

# PSEUDORANDOM NUMBER GENERATION

- วิธีนี้ใช้วิธีเดียวกับที่พวก “random number” generators ส่วนใหญ่ใช้
- โดยใช้ฟังก์ชัน ดังต่อไปนี้

$$\text{position} = (a * \text{key} + c) \bmod m$$

โดย  $a$ ,  $c$  และ  $m$  เป็นตัวเลขที่เลือกขึ้นมาเพื่อให้ผลลัพธ์

ออกมาเป็น random

ตัวเลขที่เลือกมาควรจะเป็น prime numbers ทั้งหมด

# PSEUDORANDOM NUMBER GENERATION

ตัวอย่าง: ให้  $a = 17, c = 7, m = 101$

Keys		results
123456	$(17 * 123456 + 7) \bmod 101$	80
348572	$(17 * 348572 + 7) \bmod 101$	61
298476	$(17 * 298476 + 7) \bmod 101$	61
340857	$(17 * 340857 + 7) \bmod 101$	4

# DIGIT EXTRACTION

- เลือกตัวเลขหรือกลุ่มของตัวเลขจาก key และใช้เป็น hash position.
- ตัวอย่าง
  - key เป็นตัวเลข 6 หลัก
  - เราสามารถดึงตัวเลขตำแหน่งที่หนึ่ง สามและห้า เพื่อให้เหลือตัวเลขสามหลักซึ่งสามารถเก็บในตาราง 1000 ช่อง

Keys	Digit extraction	Key mod 101
123456	135	34
348572	387	21
298476	287	21
340857	305	83

# MIDSQUARE

- นำค่า key มายกกำลังสอง แล้วเลือกตัวเลขบางตัวจากผลที่ได้ (โดยปกติจะเลือกกลุ่มตัวเลขที่อยู่ตรงกลาง) เพื่อใช้เป็น hash position.
  - ค่า key ไม่ควรเกิน 4 หลักเนื่องจากเมื่อทำการยกกำลังแล้วอาจจะมีขนาดเกินขนาดของ 32-bit integer.
- ตัวอย่าง
  - 6-digit keys.
  - เลือกตัวเลขตรงกลางสี่ตำแหน่งเพื่อนำมายกกำลังสอง และเลือกสี่ตำแหน่งกลางจากผลที่ได้เพื่อเป็น hash position

# Midsquare

Keys	Take the key	Square the result	Pick the digits
123456	2345	$2345 * 2345 = 05499025$	4990
348572	4857	$4857 * 4857 = 23590449$	5904
298476	9847	$9847 * 9847 = 96963409$	9634
340857	4085	$4085 * 4085 = 16687225$	6872

# FOLDING

- นำ key มาแบ่งเป็นกลุ่มของตัวเลขที่มีจำนวนหลักเท่าๆ กัน
- นำตัวเลขของทุกกลุ่มมาบวกเข้าด้วยกัน
- วิธีการ folding มี 2 วิธี
  - shift folding : เอา key มาแบ่งเป็นกลุ่มของตัวเลขที่มีขนาดเท่ากับขนาดของ position ที่ต้องการ แล้วนำค่าของแต่ละกลุ่มมาบวกกัน ถ้าผลที่ได้มีขนาดเกิน ให้ตัดเลขตัวหน้าทิ้ง
  - boundary folding : เมื่อแบ่งกลุ่มแล้ว นำตัวเลขที่ขอบด้านซ้ายและขวาพับมาหรือสลับกันก่อนที่จะบวกแต่ละกลุ่ม

# FOLDING

- ตัวอย่าง แบ่งเป็นกลุ่มละสองหลัก

Keys	Shift Folding	Boundary Folding
123456	$12 + 34 + 56 = 102 \rightarrow 02$	$12 + 43 + 56 = 111 \rightarrow 11$
348572	$34 + 85 + 72 = 191 \rightarrow 91$	$34 + 58 + 72 = 164 \rightarrow 64$
298476	$29 + 84 + 76 = 189 \rightarrow 89$	$29 + 48 + 76 = 153 \rightarrow 53$
340857	$34 + 08 + 57 = 99 \rightarrow 99$	$34 + 80 + 57 = 171 \rightarrow 71$



# ROTATION

- นำตัวเลข  $n$  ตำแหน่ง จากด้านหน้าหรือด้านหลังของ key แล้วหมุนไปยังด้านอีกด้านหนึ่ง
- โดยปกติแล้ว จะตามด้วยการทำ modulo division เพื่อให้ผลลัพธ์อยู่ในขอบเขตของอะเรย์ที่ใช้เก็บ
- ตัวอย่าง ( $n = 2$ )

Keys	Front --> rear	Rear --> front
123456	345612	561234
348572	857234	723485
298476	847629	762984
340857	085734	573408

# COLLISION

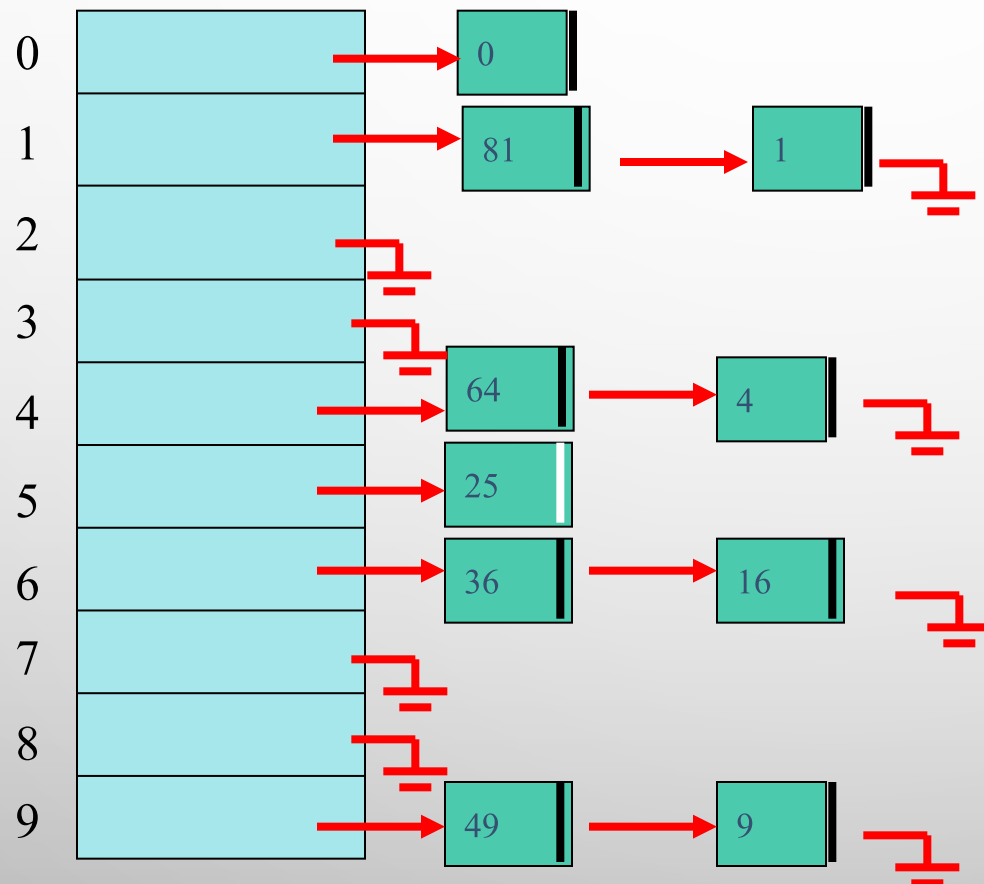
- เกิดเมื่อ keys ที่แตกต่างกันให้ค่า hashing position เดียวกัน
- เช่น ถ้าเราใช้ hash table ที่มีขนาด 101 ช่อง และใช้วิธี modulo division ในการ hash คีย์ที่มีขนาดตัวเลข 6 หลัก เช่น 348572 และ 298476 จะให้ค่า hashing position เดียวกันคือ 21
- ค่า keys ไม่สามารถเก็บไว้ในช่องเดียวกันได้

# การแก้ปัญหา COLLISION

- การแก้ปัญหา Collision จะทำโดยหาตำแหน่งใหม่ให้กับ key ถ้าตำแหน่ง hash position ของ key นั้นมีข้อมูลอยู่แล้ว
- สามารถแก้ปัญหาได้หลายวิธี เช่น
  - Separate Chaining
  - Open Addressing
    - Linear probe
    - Quadratic probe
    - Double hashing

# Separate Chaining

สร้างลิงค์ลิสต์เพื่อเก็บค่าของข้อมูลที่มี hash position เดียวกัน



สมมติ :

- hashing function คือ  $\text{hash}(x) = x \bmod 10$ .

# OPERATIONS ON SEPARATE CHAINING

- ค้นหาข้อมูล : ใช้ hash function ในการหาว่าควรจะเริ่มค้นหาที่ลิสต์ใด
- การแทรกข้อมูล (insert):
  - ค้นหาก่อนว่าข้อมูลนั้นมีอยู่แล้วในลิสต์หรือไม่
  - ถ้าไม่มี ค่อยแทรกเข้าไปในลิสต์นั้น

# INSERTION FUNCTION

กำหนดให้

TABLESIZE ขนาดของตาราง

KEYTYPE ชนิดของ key

h (KEYTYPE key) : hash function

getnode : ฟังก์ชันสำหรับขอเนื้อที่สำหรับโหนดใหม่

bucket : อะเรย์ที่เก็บ pointers ที่ชี้ไปยังลิสต์ของข้อมูล

แต่ละโหนดมีข้อมูล 2 อย่างคือ

- k : key
- next : pointer ไปยังโหนดถัดไปในลิสต์

# Insertion Function

```
struct nodetype *insert (KEYTYPE key)
{
    struct nodetype *p, *q, *s;
    i = h(key);
    for (p = bucket[i]; p != NULL && p->k != key; p = p->next;);
    if (p->k == key) return(p); /* key is already exist, do nothing */
    /* insert a new record */
    s = getnode( );
    s->k = key;
    s->next = bucket[i];
    bucket[i] = s;
    return(s);
}
```

# INSERTION FUNCTION

- สามารถแทรกโหนดที่หัวของลิสต์ เนื่องจากง่าย และข้อมูลที่เพิ่งแทรก มักจะมีโอกาสถูกเข้าถึงในอนาคตอันใกล้
- อาจมีการเรียงลำดับข้อมูลในลิสต์ใหม่เพื่อประสิทธิภาพของการค้นหา
- เวลาที่ใช้ในการค้นหาแล้วไม่เจอ จะลดลงถ้ามีการเรียงลำดับข้อมูลในลิสต์ คือเหลือประมาณครึ่งหนึ่งโดยเฉลี่ย
- ข้อเสียของวิธีนี้คือ เปลืองที่สำหรับ pointers และตารางมีขนาดใหญ่



# LOAD FACTOR

- Load factor เป็นค่าที่วัดว่าตารางใกล้เต็มแล้วหรือยัง

$$\text{Load Factor} = \frac{\text{จำนวนข้อมูลในตาราง}}{\text{ขนาดของตาราง}}$$

# OPEN ADDRESSING

- Open addressing เป็นอีกวิธีที่แก้ปัญหา collisions ด้วย linked list.
- ไม่มีการขยายขนาดของตาราง
- เมื่อเกิด collision จะทำการหาช่องว่างภายในตาราง
- แบ่งวิธีเป็น

Linear probe

Quadratic probe

Double hashing

# LINEAR PROBING

- เป็นเทคนิคการแก้ปัญหา collision ที่ง่ายที่สุด
- ข้อดีคือข้อมูลจะถูกเก็บใกล้กับตำแหน่งเดิม
- ถ้า hash position ถูกใช้แล้ว ก็หาตำแหน่งถัดไปเรื่อยๆ จนกว่าจะพบช่องว่าง
- ตัวอย่างเช่น การแทรก keys { 89, 18, 49, 58, 69} เข้าไปใน hash table โดยใช้ hash function คือ

$$\text{hash}(x) = x \bmod 10$$

Empty Table

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert 89

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Insert 18

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert 49

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert 58

0	49
1	58
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert 69

0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

# QUADRATIC PROBING

- วิธีนี้จะช่วยลดปัญหาการกระจุกตัวของข้อมูล (primary clustering problem) เมื่อใช้วิธี linear probing.
- ทุกครั้งที่มีการซ้ำ จะหาช่องถัดไปที่  $i^2$  โดยที่  $i$  เริ่มจาก 1 และเพิ่มขึ้นเรื่อยๆ ทีละ 1 ถ้ายังไม่พบช่องว่าง
- สมมติ hashing function คือ

$$\text{hash}(x) = x \bmod 10$$

Empty Table

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert 89

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Insert 18

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert 49

0	49
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

Insert 58

0	49
1	
2	58
3	
4	
5	
6	
7	
8	18
9	89

Insert 69

0	49
1	
2	58
3	69
4	
5	
6	
7	
8	18
9	89



# QUADRATIC PROBING (CONT.)

- วิธี linear probing ถ้าปล่อยให้ hash table โกงเต็ม ประสิทธิภาพจะลดลงมาก
- วิธี quadratic probing จะยังแย่งในสถานการณ์ดังกล่าว
- ถ้าขนาดของตารางไม่ใช่ prime อาจทำให้ไม่สามารถหาช่องว่างได้หลังจากที่ตารางเต็มประมาณครึ่งหนึ่งแล้ว
- เป็นเพราะครึ่งหนึ่งของตารางจะถูกใช้ช่องเก็บข้อมูลที่เกิดจากการ collisions.
- ถ้าตารางว่างครึ่งหนึ่งและขนาดเป็นเลข prime จึงจะรับประกันได้ว่าจะสามารถหาที่ว่างสำหรับข้อมูลใหม่ได้

# DOUBLE HASHING

- ถ้าเกิด collision ก็ทำ hashing อีกครั้งหนึ่ง ส่วนมากจะใช้ second hash function
- ถ้าเลือก  $\text{hash}_2(x)$  ไม่ดีก็จะยิ่งแย่ไปใหญ่
- Hash function ต้องไม่ให้ค่า 0
- Hash function ที่นิยมใช้ครั้งที่สองคือ

$$\text{hash}_2(x) = R - (x \bmod R)$$

- R เป็น prime ที่มีขนาดเล็กกว่า TableSize.

สมมติ : hashing function คือ  $\text{hash}(x) = x \bmod 10$

Empty Table

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Insert 89

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	89

Insert 18

0	
1	
2	
3	
4	
5	
6	
7	
8	18
9	89

$$\text{hash}_2(x) = R - (x \bmod R), \quad R = 7$$

Insert 49

0	
1	
2	
3	
4	
5	
6	49
7	
8	18
9	89

Insert 58

0	
1	
2	
3	58
4	
5	
6	49
7	
8	18
9	89

Insert 69

0	69
1	
2	
3	58
4	
5	
6	49
7	
8	18
9	89

# REHASHING

- ถ้าค่า load factor มากเกินไป แสดงว่าตารางใกล้เต็มแล้ว อาจใช้เทคนิค rehashing เพื่อแก้ปัญหา ทำโดย
  - สร้างตารางใหม่ที่มีขนาดเป็นสองเท่าของของเดิม และใช้ hash function ใหม่สำหรับตารางใหม่
  - เอาค่าในตารางเดิมมาทำ hashing ใหม่ เพื่อหา hash position ในตารางใหม่
- การทำ Rehashing สามารถทำได้หลายวิธี

ตัวอย่าง

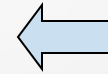
- แทรกค่า 13, 15, 24, 6
- hash function คือ  $h(x) = x \bmod 7$
- Linear probing ถูกใช้ในการแก้ปัญหา collisions.

0	6
1	15
2	
3	24
4	
5	
6	13

Load factor = 0.57

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

Load factor = 0.71



- หลังจาก 23 ถูก insert ตารางจะมีข้อมูลเกิน 70%
- ตารางมีข้อมูลมากเกินไป จึงต้องสร้างตารางใหม่
- ขนาดขงตารางใหม่ควรจะใหญ่เป็นสองเท่าของตารางเดิม และควรเป็น prime (17)
- hashing function ใหม่คือ
$$h(x) = x \bmod 17$$
- อ่านข้อมูลตารางเก่าทีละช่อง หา hash position และใส่ในตารางใหม่

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



## เมื่อไหร่ต้องทำ REHASHING

- ☐ เมื่อตารางเต็มครึ่งหนึ่งแล้ว
- ☐ เมื่อไม่สามารถแทรกข้อมูลใหม่ได้แล้ว
- ☐ เมื่อค่า **load factor** ของตารางเกินค่าที่กำหนดไว้

# STRING HASHING

- หาค่า ASCII ของแต่ละตัวอักษรเพื่อแปลงเป็นตัวเลข
- นำค่าที่แปลงได้ของแต่ละตัวมาพับรวมกัน หรือบวกกันเพื่อให้ได้
- หลักของตัวเลขน้อยลง แล้วจึงนำไปใช้กับ hash function
- ตัวอย่างการ hash ข้อมูล “Hello”.
  - Hello มีลำดับค่า ASCII คือ 72, 101, 108, 108, 111
  - พับหรือบวกชุดของตัวเลขเข้าด้วยกัน ได้ค่า 500
  - นำค่านี้เป็นจุดเริ่มต้นเพื่อใช้กับ hashing function ที่เลือก

# HASHING กับ BINARY SEARCH TREE

	Binary search tree	Hash table
<input type="checkbox"/> <i>insert</i> and <i>find</i> operation.	Yes	Fastest
<input type="checkbox"/> Support routine that require order	Powerful	-
<input type="checkbox"/> Find minimum/ maximum element.	Yes	-
<input type="checkbox"/> Quickly find all item in a certain range	Yes	-
<input type="checkbox"/> Worst case	Sorted Input	Implementation Error