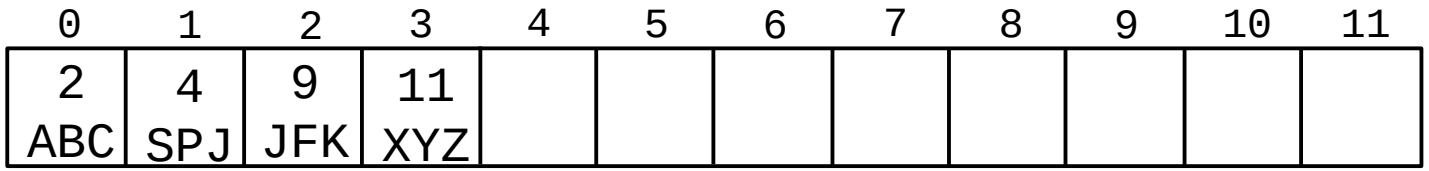
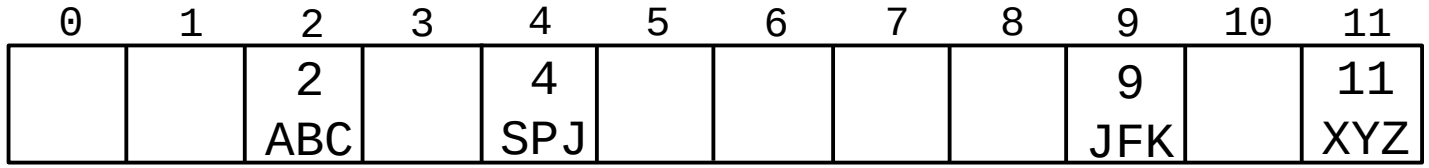
TABLES & INFORMATION RETRIEVAL

การสืบค้นข้อมูล (Information Retrieval)

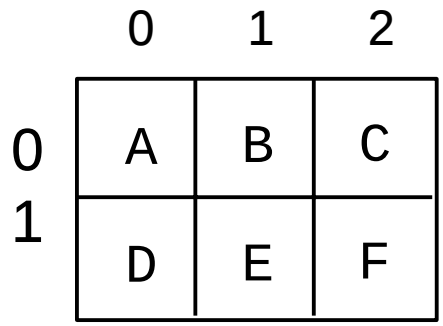
- เวลาในการสืบค้นข้อมูลที่เร็วที่สุด
  - โดยใช้วิธีการเปรียบเทียบ O(log n)



  - โดยใช้ตารางเก็บข้อมูล O(1)



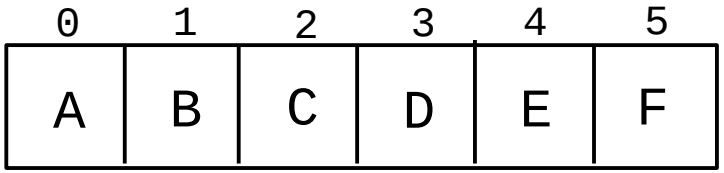
Rectangular Arrays




```
char    data[2][3];
```

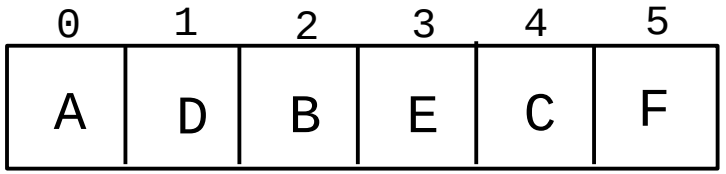
- Row-major ordering

```
A[0][0],A[0][1],A[0][2],A[1][0],A[1][1],A[1][2]
```

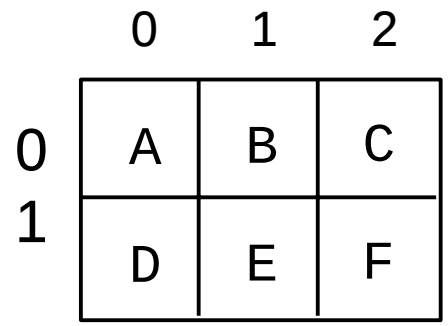


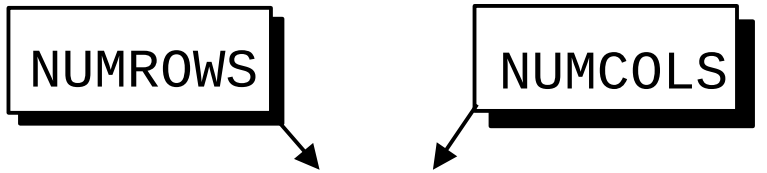
- Column-major ordering

```
A[0][0],A[1][0],A[0][1],A[1][1],A[0][2],A[1][2]
```

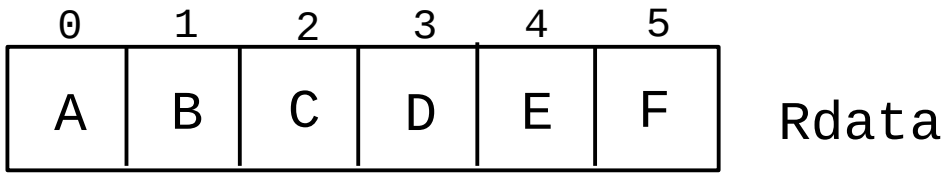


Index Functions



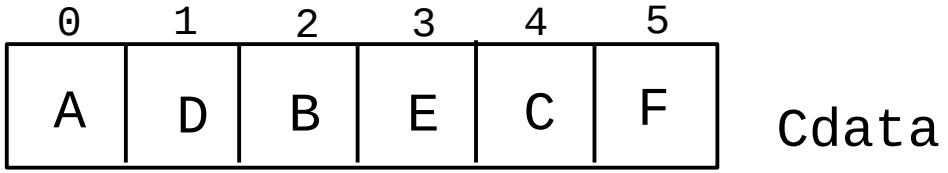



```
char    data[2][3];
```



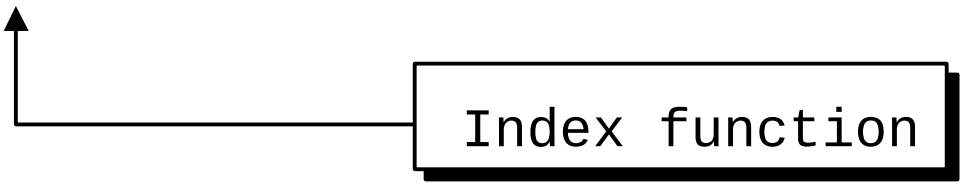
- Row-major ordering

```
Rdata[ NUMCOLS × r + c ] = data[r][c];
```

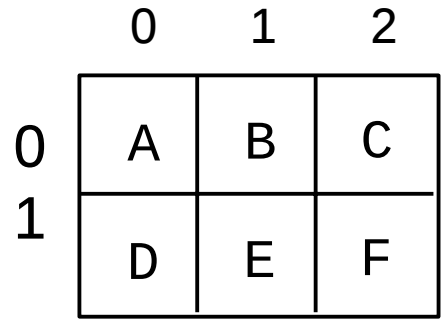


- Column-major ordering

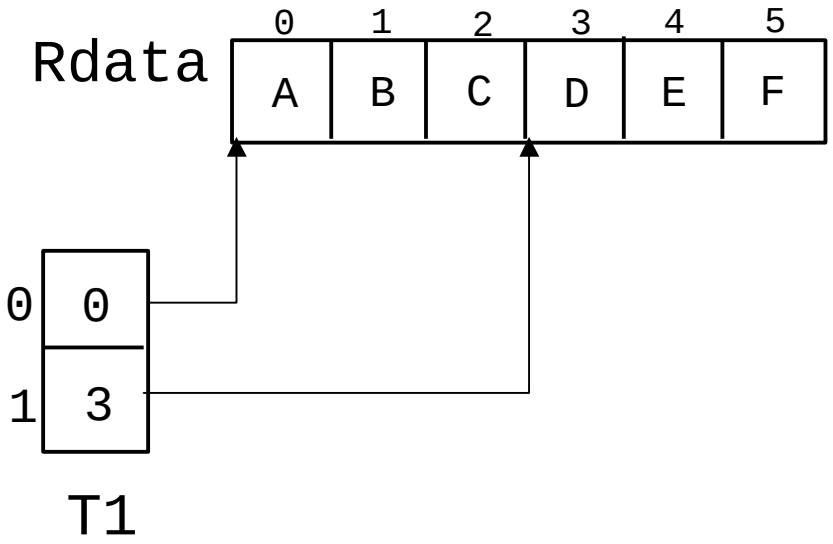
```
Cdata[ NUMROWS × c + r ] = data[r][c];
```



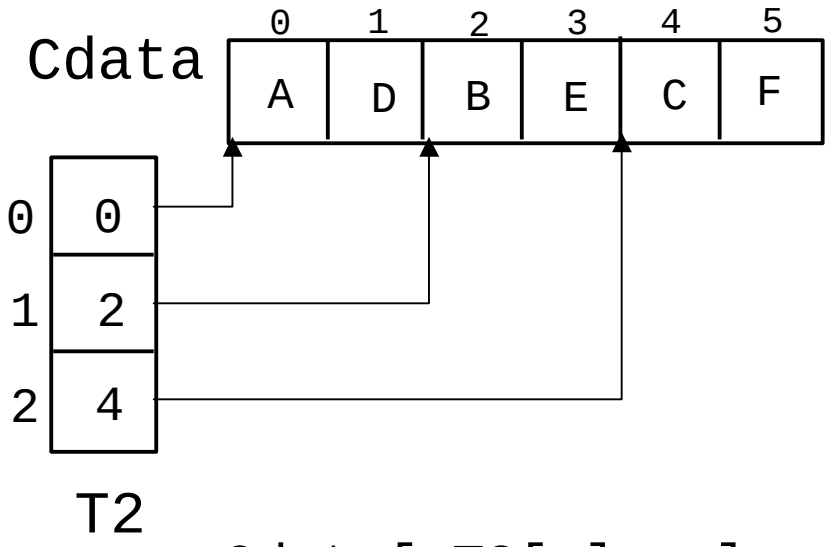
Access Tables




```
char    data[2][3];
```

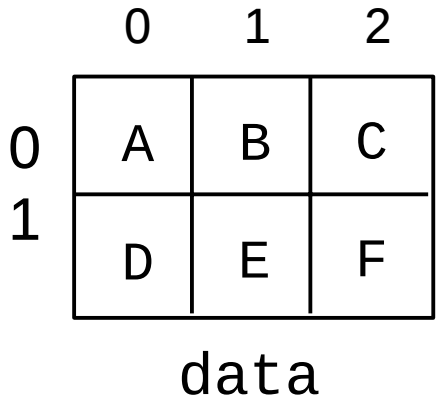


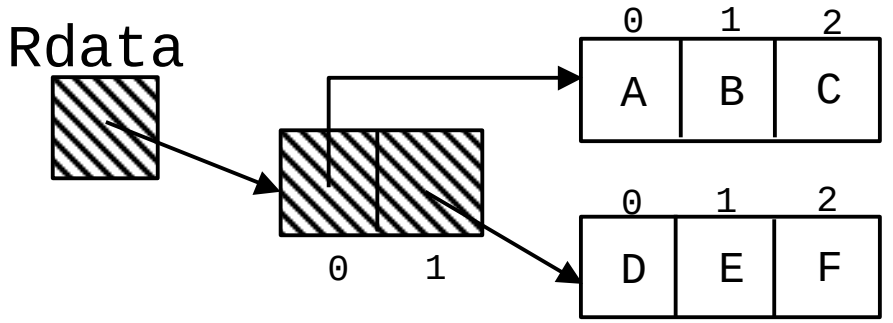

```
Rdata[ T1[r]+c ]
= data[r][c]
```




```
Cdata[ T2[c]+r ] =
data[r][c]
```

Access Tables in C

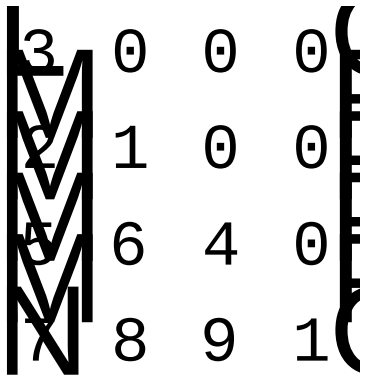


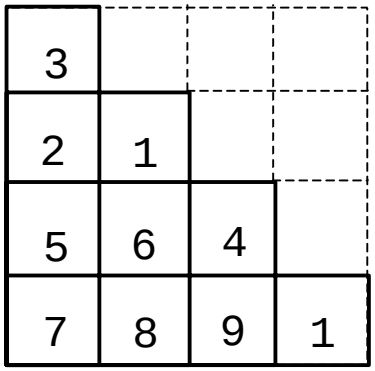


Rdata[r][c] = data[r][c]

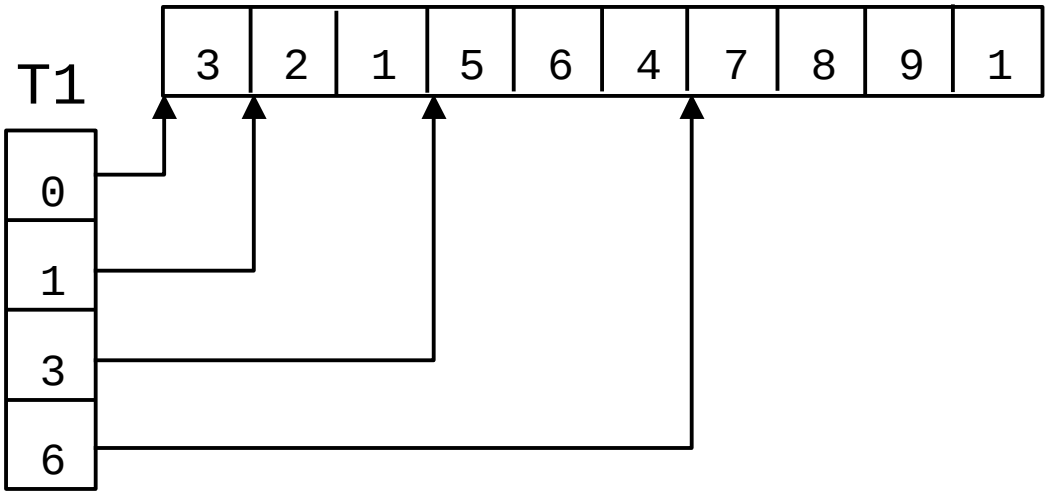
```
char    data[2][3];
char    **Rdata;

Rdata    = malloc( (unsigned) (2*sizeof(char *)) );
Rdata[0] = malloc( (unsigned) (3*sizeof(char)) );
Rdata[1] = malloc( (unsigned) (3*sizeof(char)) );
```

Triangular Tables





data[r][c]

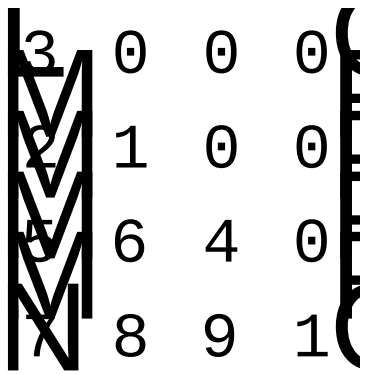


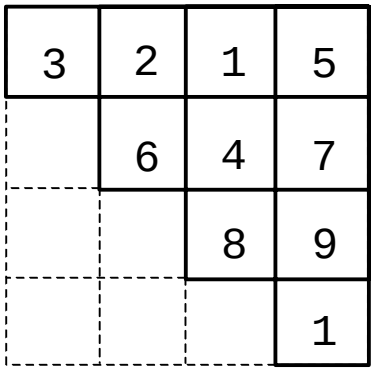
Access Table

Index function

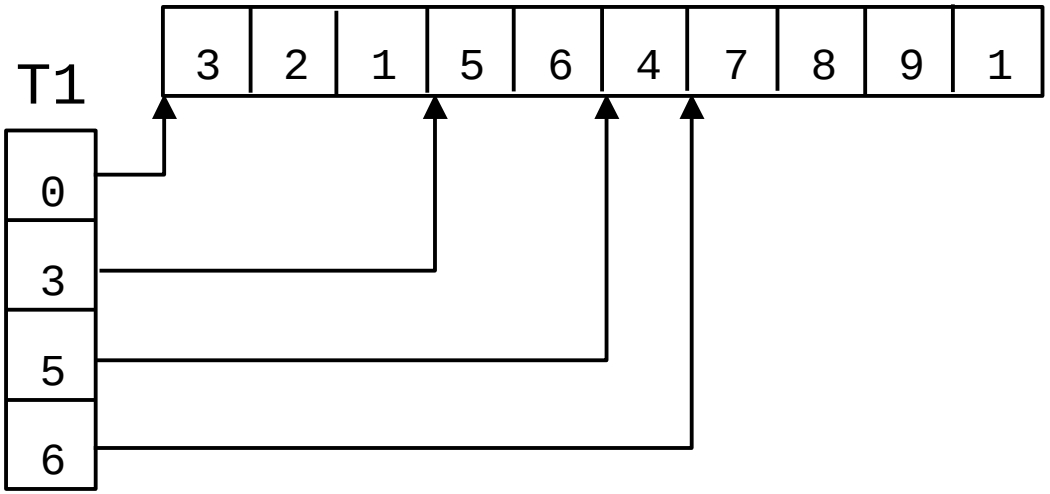
```
data[r][c] = Rdata[ T1[r]+c ]
data[r][c] = Rdata[ 0.5r(r+1) + c ]
```

Triangular Tables





data[r][c]

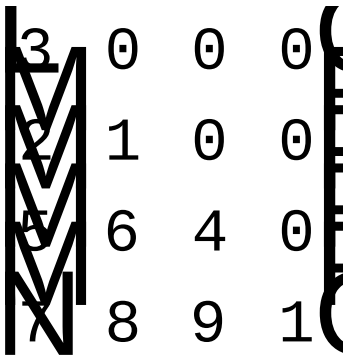


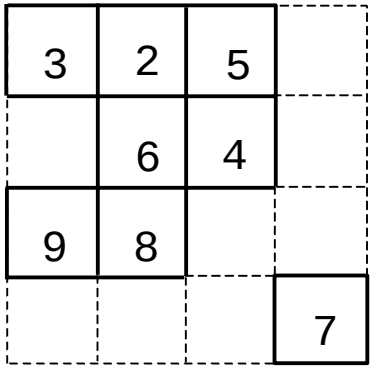
Access Table

Index function

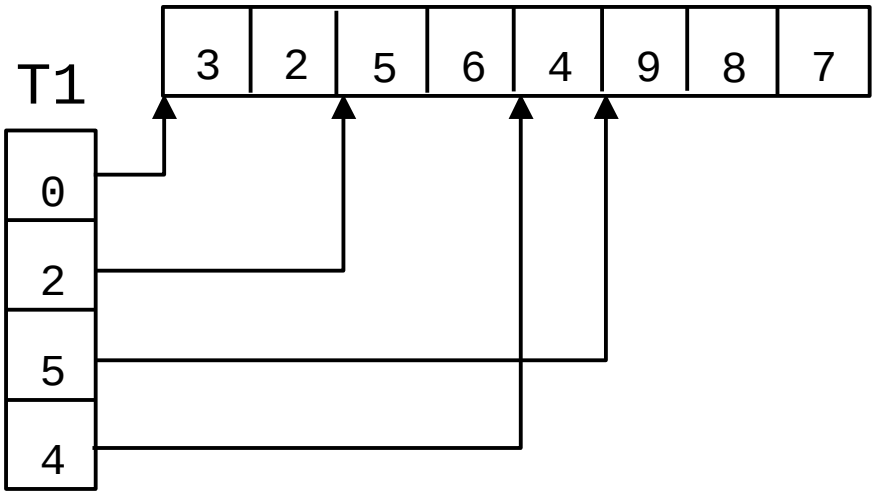
```
data[r][c] = Rdata[ T1[r]+c ]
data[r][c] = Rdata[ NUMCOLS × r + c - 0.5r(r-1) ]
```

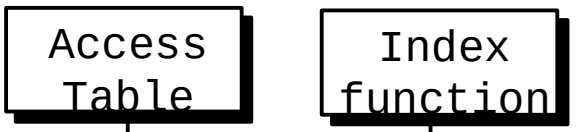
Jagged Tables





data[r][c]






```
data[r][c] = Rdata[ T1[r]+c ]
data[r][c] = Rdata[ ??? ]
```

Inverted Tables

Index	ชื่อ	IQ	ความสูง
0	ติงต๊อง	180	167
1	ตอเต่า	90	153
2	เตอร์เติล	110	169
3	โตงเตง	132	180
4	ต้อยต่ำ	50	171

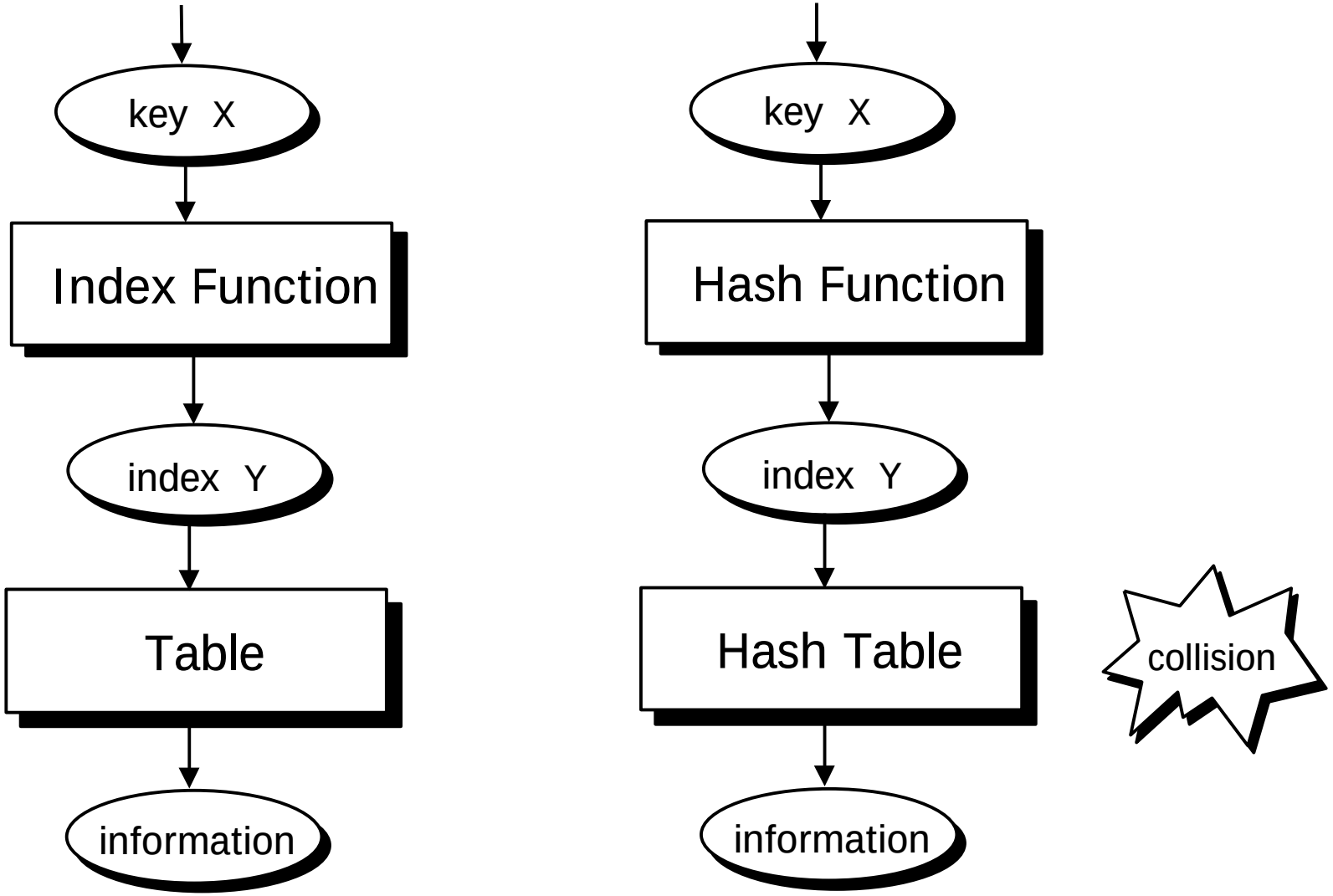
Main tables

IQ
0
3
2
1
4

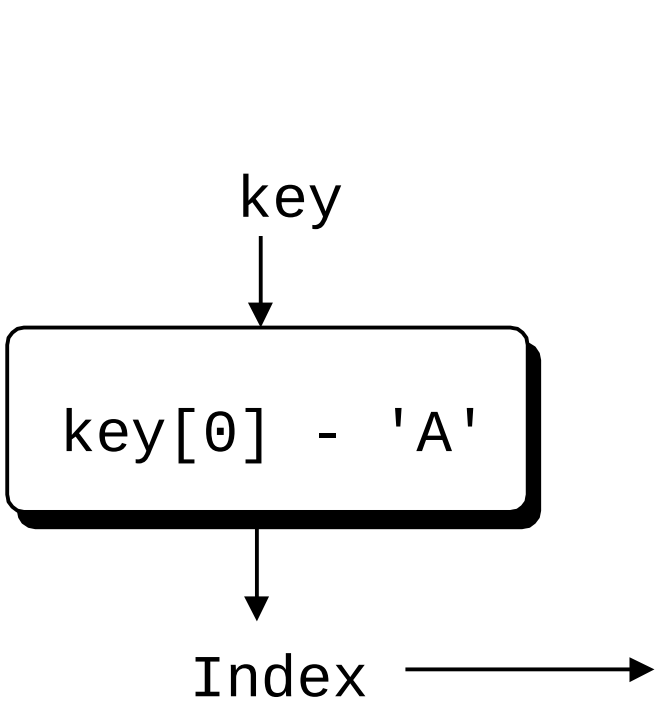
ความสูง
3
4
2
0
1

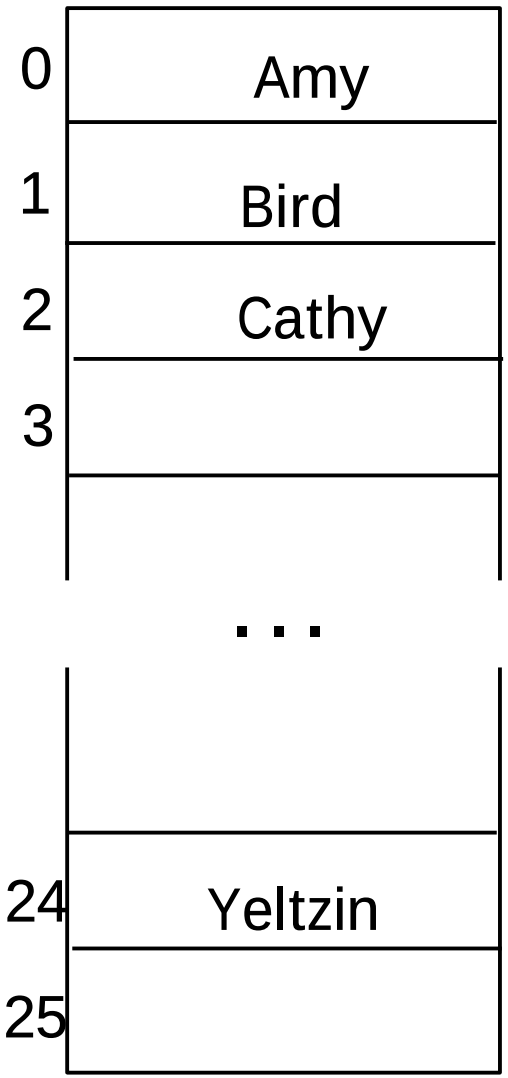
access tables

Hashing



Hashing





คีย์ที่แตกต่างกันอาจถูกเปลี่ยนเป็น index เดียวกันได้ จึงต้องมีการทำ collision resolution

Hash Functions

- สมมติให้คีย์ประกอบด้วยตัวเลข 8 หลัก
  และข้อมูลที่จะถูกเก็บมีประมาณ 1000 ตัว
- Truncation : ใช้ตัวเลขหลักที่แปด, ห้า และหนึ่ง

  h(6523489**0**) = 630
- Folding : แบ่งคีย์ออกเป็นตัวเลข 3 จำนวน รวมค่าของแต่ละจำนวนเข้าด้วยกัน แล้วปัดตัวเกินทิ้ง

  h(65234890) = 65 + 234 + 890 = 1189 = 189
- Modular Arithmetic : (ค่าของคีย์) % (prime number)

  h(65234890) = 65234890 % 997 = 183

Hash Functions

```
int  Hash( char *pKey )
{
  int     h = 0;

  while ( *pKey != NULL ) {
    h += *(pKey++);
  }
  return( abs( h % HASHSIZE );
}
```

คีย์เป็นข้อมูล ประเภท character string

ถ้ากำหนดให้ 'SOMCHAI' เป็นคีย์ และ HASHSIZE = 251

จะได้ว่า 'S' + 'O' + 'M' + 'C' + 'H' + 'A' + 'I'

= 53H + 4FH + 4DH + 43H + 48H + 41H + 49H

= 204H = 516

h = 516 % 251 = 14

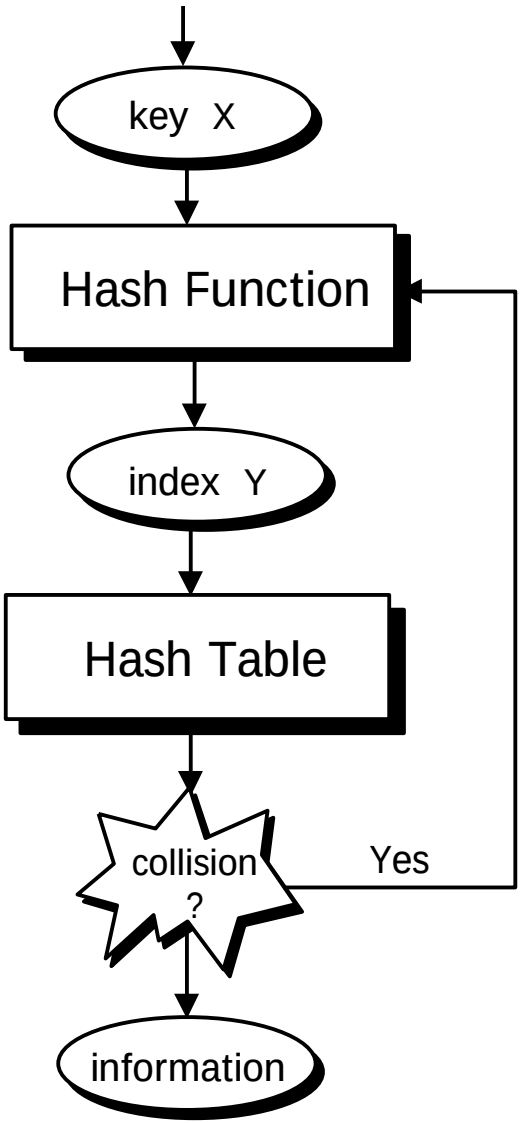
Collision Resolutions

Collision คือ กรณีที่มีคีย์มากกว่าหนึ่งคีย์ถูกเปลี่ยนไปสู่ index เดียวกัน ในตาราง สามารถแก้ไขได้โดยการเก็บข้อมูลในตารางในลักษณะดังนี้

- Open addressing
  - Rehashing
  - Linear probing
  - Quadratic probing
  - Key-dependent increment
  - Random probing
- Chaning

Rehashing

- ถ้าเกิด collision ให้เปลี่ยน hash function แล้วลองใหม่ (rehashing)
- โดยทั่วไปจะนำตำแหน่งที่เกิด collision มาเป็นตัวแปรหนึ่งในการหาตำแหน่งใหม่ ของที่เก็บข้อมูล



Linear Probing

ตำแหน่งใหม่คือตำแหน่งถัดไปในตาราง ถ้าเกิดการชนอีกก็พิจารณา ตำแหน่งถัดไปอีก ทำเช่นนี้เรื่อยๆจนกว่าจะพบตำแหน่งว่างในตาราง

นั่นคือ ตำแหน่งใหม่ที่จะพิจารณา = `(h + i) % HASHSIZE`

โดยที่ `h` คือค่าที่ได้จาก hash function

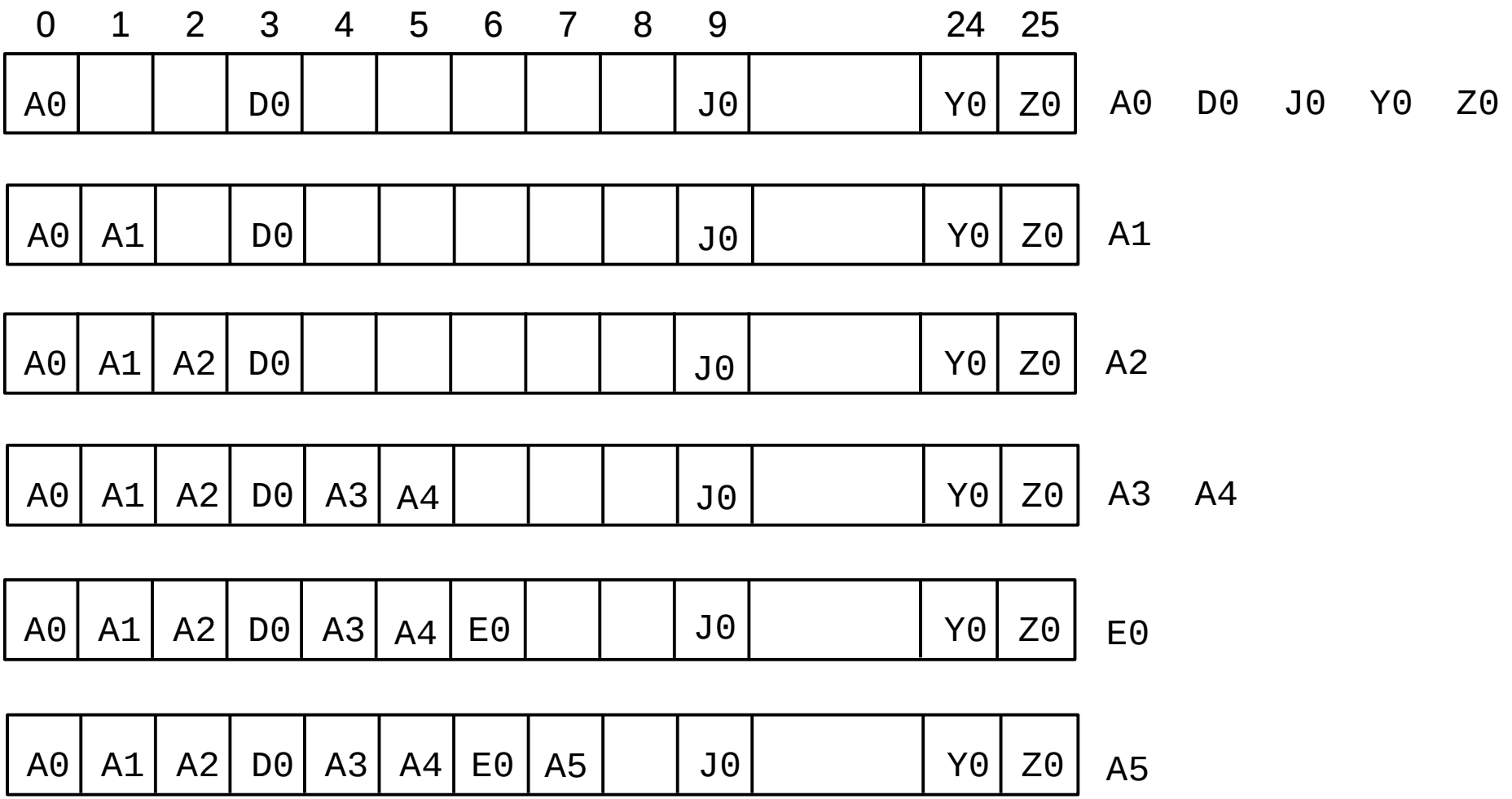
และ `i` คือครั้งที่เกิดการชน

ดังนั้น ลำดับของตำแหน่งในตารางที่ถูกพิจารณาคือ

```
h, h+1, h+2, h+3, ...
```

วิธีนี้มักทำให้เกิดการเกาะกลุ่มของข้อมูล (clustering) ทำให้การหา ตำแหน่งใหม่เมื่อเกิดการชนช้าลง

Linear Probing



hash function : `h = key[0] - 'A'`

Quadratic Probing

ลักษณะการหาตำแหน่งใหม่ในกรณีที่ชนจะคล้ายกับ linear probing โดยที่ ตำแหน่งใหม่ที่จะพิจารณา = `(h + i ) % HASHSIZE`^{2}

โดยที่ `h` คือค่าที่ได้จาก hash function

และ `i` คือครั้งที่เกิดการชน

ดังนั้น ลำดับของตำแหน่งในตารางที่ถูกพิจารณาคือ

```
h, h+1, h+4, h+9, ...
```

ถ้า `HASHSIZE` เป็นเลข prime วิธีนี้จะ probe ตำแหน่งใน table จำนวนเท่ากับ `(HASHSIZE/2)`

Key-Dependent Increment

- ในการหาตำแหน่งใหม่กรณีที่เกิดการชน ระยะที่เพิ่ม (increment) จาก ตำแหน่งที่เกิดการชน ในกรณีของ linear หรือ quadratic probing จะขึ้นกับจำนวนครั้งที่เกิดการชน
- สำหรับวิธีนี้ แนะนำให้ระยะเพิ่มนี้ ขึ้นกับค่าของคีย์ด้วย โดยระยะเพิ่มนี้ จะคงที่ไปตลอดการหาที่ว่างสำหรับคีย์นั้น
- ตัวอย่างเช่น กำหนดให้ระยะเพิ่มคือ ค่าของตัวเลขสองหลักล่างสุด ดังนั้นถ้าคีย์มีค่า 65234890 จะมีระยะเพิ่มเป็น 90
- ดังนั้นถ้า HASHSIZE เป็นเลข prime จะต้องพบที่ว่างในตาราง

Quadratic Probing

```
void  Insert( HTableType H, ItemType r )
{
  int    c = 0,  i = 1, p;

  p = Hash( r.key );
  while ( H[p].key != NULL &&
          NE( r.key, H[p].key ) && c <=HASHSIZE/2 ) {
    c++;
    p += i;  i += 2;
    p = p % HASHSIZE;
  }
  if ( H[p].key == NULL )
```

H[p].key = r;

else if (EQ(r.key, H[p].key)

Error("Duplicate key");

else

Collision Resolution by Chaining

ข้อด

- ประหยัดเนื้อที่ในกรณีที่ record มีขนาดใหญ่
- ง่ายต่อการทำ collision resolution
- ขนาดของ hash table ไม่จำเป็นต้องถูกจองเกินกว่าจำนวนข้อมูลที่จะเก็บ
- ง่ายเมื่อต้องการลบข้อมูล

(open addressing ลบลำบาก ???)

ข้อเสีย :

- link กินเนื้อที่

Analysis of Chaining

- ให้ t คือขนาดของ hash table
- ให้ n คือจำนวนข้อมูลที่มีอยู่ใน table
- Load facter $L = n/t$
- จำนวนครั้งเฉลี่ยในการเปรียบเทียบข้อมูล (probing)
  - ในกรณีที่หาข้อมูลไม่พบ

    L
  - ในกรณีที่หาข้อมูลพบ

    $0.5(1 + (n-1)/t + 1) \approx 1 + 0.5L$

Analysis of Hashing

	จำนวนครั้งเฉลี่ยในการเปรียบเทียบ	
	กรณีหาไม่พบ	กรณีที่หาพบ
Chaining	L	$1+0.5L$
Random probe	$1/(1-L)$	$-(1/L)\ln(1-L)$
Linear probe	$0.5[1 + 1/(1-L)_2]$	$0.5[1+1/(1-L)]$