ตารางแฮช

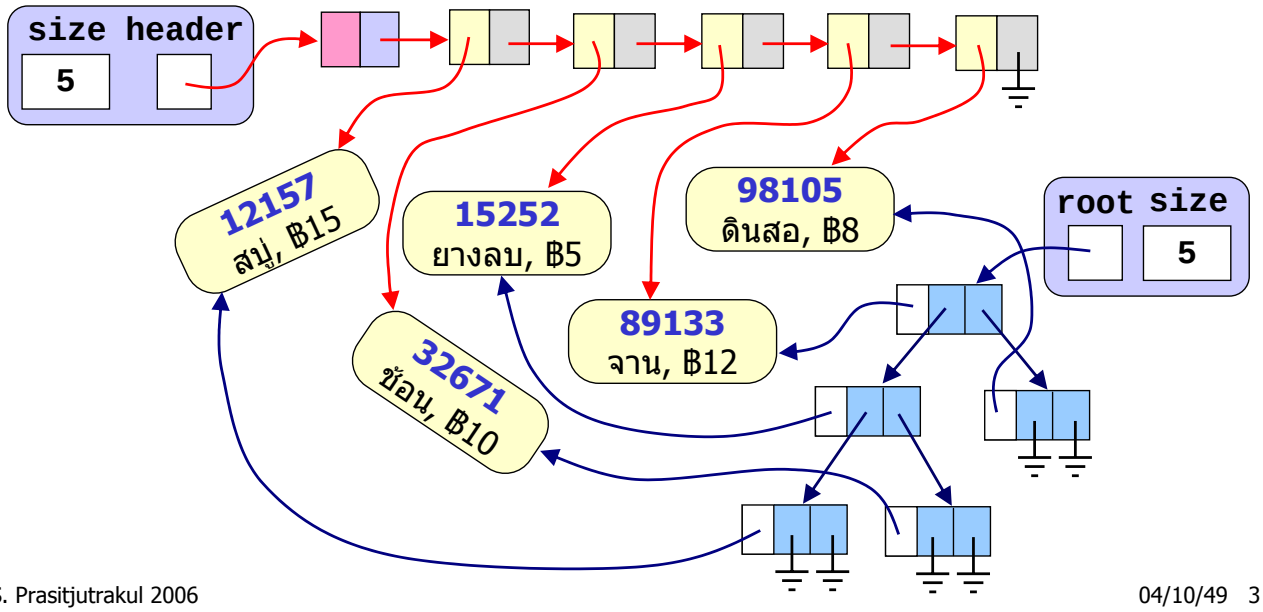
(Hash Tables)

หัวข้อ

- การใช้ตารางเก็บข้อมูลด้วยฟังก์ชันดัชนี
- การเก็บข้อมูลแบบแยกกันโยง
- ฟังก์ชันแฮช
- กลวิธีการเขียนฟังก์ชันแฮช
- การแฮชในจาวา
- การกำหนดเลขที่อยู่เปิด
- การเกาะกลุ่มของข้อมูล

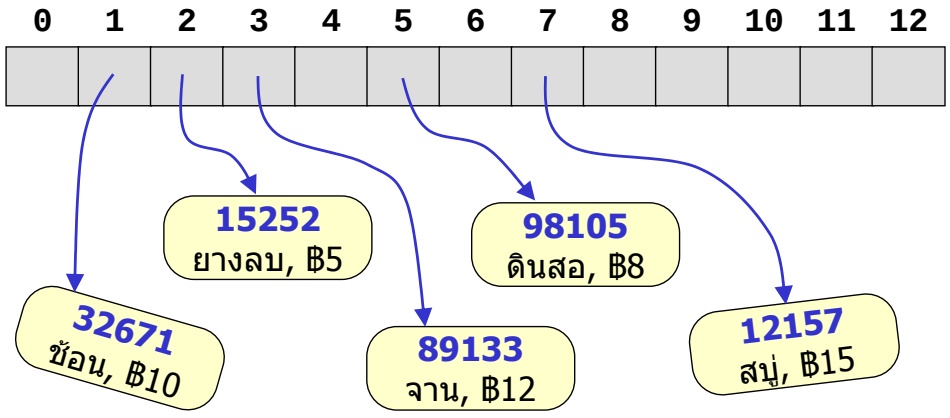
ที่เก็บข้อมูล

- เก็บในรายการ (list) : O(n)
- เก็บในต้นไม้เอวีแอล : O(log n)
- ทำอย่างไรให้เร็วกว่านี้ ?



ใช้ฟังก์ชันดัชนีคำนวณตำแหน่ง

- key ของข้อมูลคือส่วนของข้อมูลที่ใช้ในการค้น
- มีตารางซึ่งแต่ละช่องเป็นที่เก็บข้อมูล
- หา f(key) เพื่อแปลง key ไปเป็น index ของตาราง
- ฟังก์ชันดัชนีหาไม่ยาก ถ้าจองตารางขนาดใหญ่ ๆ



f(key) = key % 10

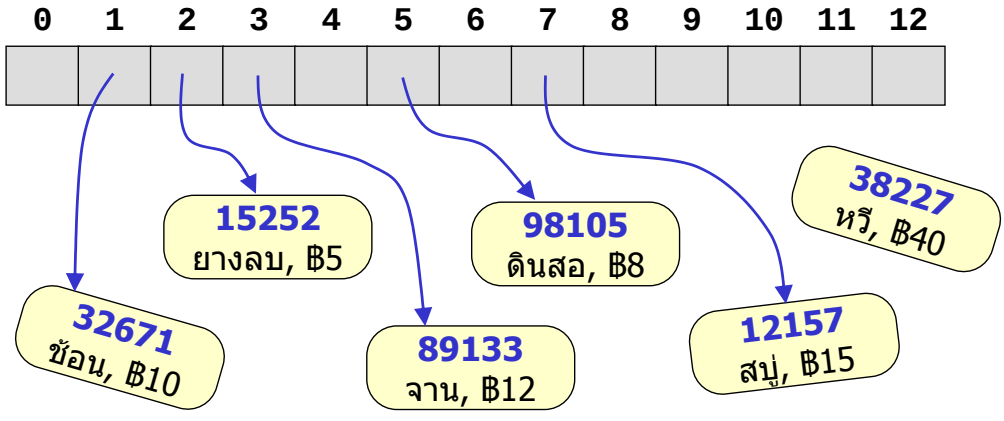
Table

```
public class Table implements Set {
  private Object[] table;
  private int size = 0;

  public Table(int m) { table = new Object[m]; }
  public boolean isEmpty() { return size == 0; }
  public int size() { return size; }

  public void add(Object x) {
    if (table[f(x)] == null)
      {++size; table[f(x)] = x;}
  }
  public void remove(Object x) {
    if (table[f(x)] != null && table[f(x)].equals(x))
      {--size; table[f(x)] = null;}
  }
  public void contains(Object x) {
    return table[f(x)] != null && table[f(x)].equals(x);
  }
  private int f(Object x) { ... }
}
```

$\Theta(1)$

ฟังก์ชันดัชนีนั้นหายาก

- เมื่อต้องเก็บอย่างประหยัด
- เมื่อต้องประกันว่าไม่เกิดการ "ชน"
- ถ้ารู้ชุดข้อมูลที่จะจัดเก็บก่อน ก็อาจหาสูตรที่ไม่ชนได้
- แต่ในทางปฏิบัติ ไม่รู้



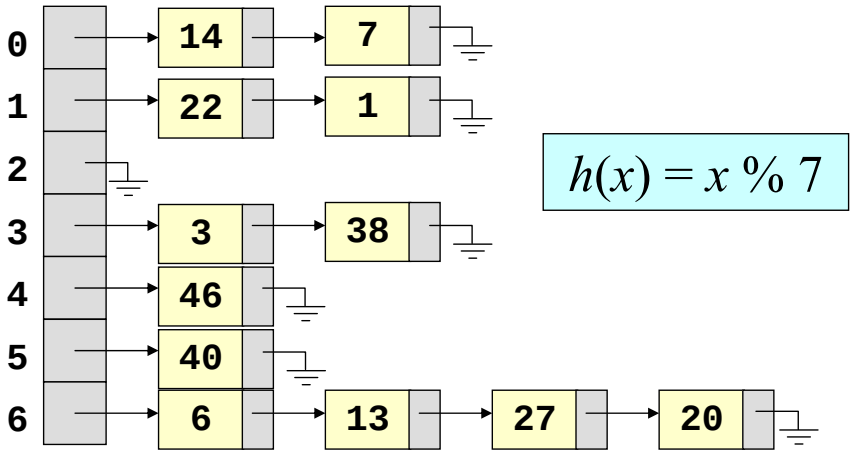
f(key) = key % 10

เปลี่ยนกลยุทธ์ : อนุญาตให้ชนได้

- จะได้เก็บข้อมูลในตารางที่ไม่ใหญ่มาก
- แต่ต้องหาวิธีแก้ไขปัญหาการชน ที่ทำงานได้เร็ว ๆ

Separate Chaining

- จัดเก็บกลุ่มข้อมูลที่ชนกันไว้ในรายการเดียวกัน



SeparateChaining

```
public class SeparateChaining {
  private LinkedList[] table;
  private int size = 0;

  public SeparateChaining(int m) {
    LinkedList[] table = new LinkedList[m];
    for (int i=0; i<table.length; i++)
      table[i] = new LinkedList();
  }
  public int size() {
    return size;
  }
  public boolean isEmpty() {
    return size == 0;
  }
  ...
```

SeparateChaining

```
public class SeparateChaining {
  ...
  public boolean contains(Object x) {
    return table[h(x)].contains(x);
  }
  public void add(Object x) {
    table[h(x)].add(0, x);
    ++size;
  }
  public void remove(Object x) {
    int i = h(x);
    int s = table[i].size();
    table[i].remove(x);
    if (s > table[i].size()) size--;
  }
  private int h(Object x) {  ...  }
}
```

add ใช้เวลาคงตัว

contains และ remove ใช้เวลาแปรตามความยาว list

การกระจายของข้อมูล

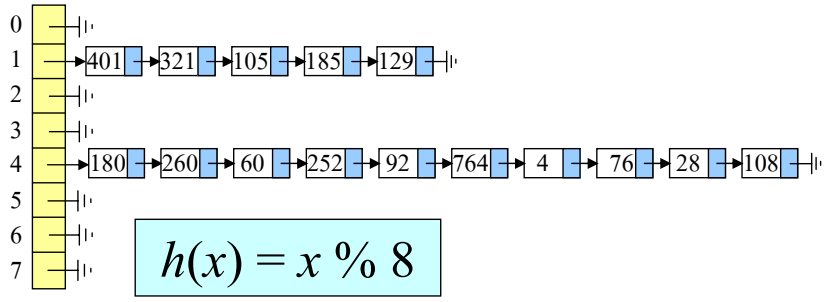
- ถ้าข้อมูลกระจายทั่วตาราง
  - แต่ละช่องเก็บรายการยาว ≈ λ
  - ถ้า λ น้อย ค้นหาได้เร็ว
- ถ้าไม่กระจาย
  - มีบางรายการยาวเกิน λ มาก
  - การค้นหาช้าเหมือนเก็บด้วย list

load factor

$$\lambda = n / m$$

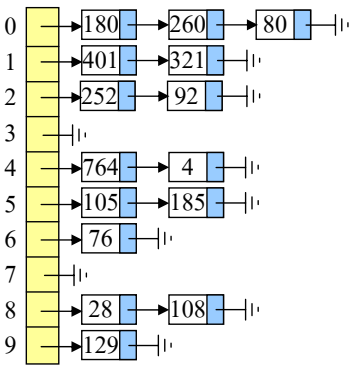
ปริมาณข้อมูล

ขนาดของตาราง



$h(x) = x \% 8$

$h(x) = x \% 10$

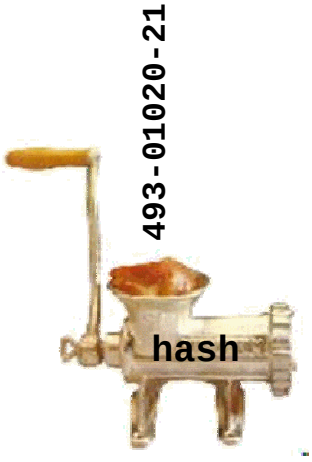


การกระจายของข้อมูล

- ขึ้นกับ
  - x : คีย์ของข้อมูล
  - h(x) : ฟังก์ชันการแปลงคีย์เป็นเลขที่ช่องของตาราง
- ถ้ากลุ่มข้อมูลมีคีย์ x ที่มีค่ากระจายอยู่แล้ว
  - ถ้าใช้ตาราง 100 ช่อง, ก็ให้ h(x) = x % 100
  - ถ้าใช้ตาราง 2^k ช่อง, ก็ให้ h(x) = k บิตทางขวาของ x
- ถ้ากลุ่มข้อมูลมีคีย์ที่มีค่าเป็นระเบียบ
  - รหัสนักศึกษา, รหัสประจำตัวบัตรประชาชน, ...
  - ต้องออกแบบ h(x) ให้ทำ x ที่มีระเบียบให้ "เละ"
  - เรียก h(x) ว่าฟังก์ชันแฮช (Hash function)

ฟังก์ชันแฮช (Hash Function)

- www.webster.com
  - hash : to chop (as meat and potatoes) into small pieces
- สอ เสถบุตร
  - สับ, แหลก, นำมาโขลกเข้าด้วยกัน




```
493-01020-21  →  10291
493-87628-21  →  76102
473-12332-21  →  40001
463-09872-21  →  00012
```

ตัวอย่างฟังก์ชันแฮช

```
static int h1(int x) {
  long hash = (2654435769L * x) & 0xFFFFFFFFL;
  return (int) (hash >> 22);
}
```

```
static int h2(int x) {
  x = ~x + (x << 15);
  x ^= (x >>> 11);
  x += (x << 3);
  x ^= (x >>> 5);
  x += (x << 10);
  x ^= (x >>> 16);
  return x;
}
```

x	1	2	3	4	5	6	7	8
h1(x)	632	241	874	483	92	725	334	966
h2(x)	500	1001	507	978	486	1014	403	933

กลวิธีการเขียนฟังก์ชันแฮช

- การวิเคราะห์เลขโดด (digit analysis)
- การคูณ (multiplicative hashing)
- การพับ (folding)
- การหาร (modulus hashing)

การวิเคราะห์เลขโดด (Digit Analysis)

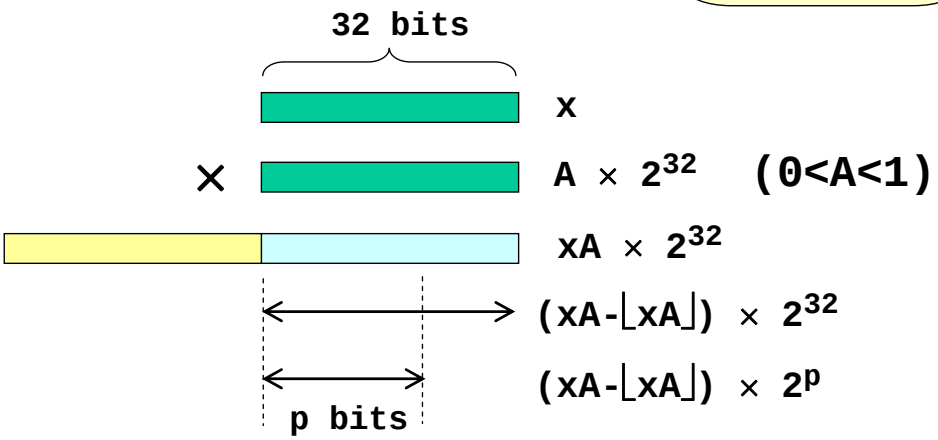
- คัดเลือกเลขโดดบางหลักของคีย์มาพิจารณา
- มั่นใจว่าที่ตัดไปไม่ทำให้เกิดความเอนเอียงในการกระจายของคีย์
- เช่น
  - รหัสนิสิตวิศวฯ ป.ตรี มีรูปแบบ : xx3xxxxx21
  - ก็ตัดเลข 3 และ 21 ออกจากการพิจารณา
  - k = 4830109521,
  - k1 = $\lfloor k / 100 \rfloor$ // k1 = 48301095
  - k2 = $\lfloor k1 / 10^6 \rfloor$ // k2 = 48
  - k3 = $k2*10^5 + k1 \% 10^5$ // k3 = 4801095

การคูณ (Multiplicative Hashing)

- คูณคีย์ด้วยจำนวนจริง A ที่มีค่าระหว่าง (0,1)
- นำเศษมาคูณกับขนาดของตาราง (m = 2^p)

$$h(x) = \lfloor m\left(xA - \lfloor xA \rfloor\right) \rfloor$$

ส่วนที่เป็นเศษของ xA



การคูณ : Fibonacci Hashing

- ถ้า A = golden ratio 0.6180339887... $\hat{\phi} = \frac{\sqrt{5}-1}{2}$
  จะแยกคีย์ที่มีค่าใกล้กันออกจากกันได้ดี

```
int multHash(int x, int p) {
  long s = 2654435769L;
  long hash = (s * x) & 0xFFFFFFFFL;
  return (hash >> (32-p));
}
```

$0.6180339887 \times 2^{32}$

```
for (int i = 0; i < 10; i++) {
  System.out.print(multHash(i, 16)+",");
}
```

```
0,40503,15470,55974,30941,5909,46412,21380,61883,36851
```


การพับ (Folding)

- แบ่งคีย์ออกเป็นส่วนๆ แล้วนำมา "รวม" กัน
- "รวม" ≡ บวก, xor, ...

```
2 1 0 2 9 3 8 4 5 0 5 0

      9 3 8 4
+     2 1 0 2
      5 0 5 0
-------------
    1 6 5 3 6
=============
```

การหาร (Modulus Hashing)

- $h(x) = x \% p$
- ไม่ควรเลือก
  - $p = 10^q$ เพราะเลือกเฉพาะ q หลักขวา ถ้าคีย์เป็นฐานสิบ
  - $p = 2^q$ เพราะเลือกเฉพาะ q บิตขวา
  - p ที่มีค่าน้อย ๆ เป็นตัวประกอบ
    - ถ้า c คือตัวประกอบร่วมของ p และ x
    - ค่า $x \% p$ จะเป็นจำนวนเท่าของ c
    - ถ้า c มีค่าน้อย ๆ จะมีคีย์จำนวนมากที่ได้ $x \% p$ มีค่าเป็นจำนวนเท่าของตัวประกอบนั้น ซึ่งไม่กระจาย
- โดยทั่วไปเลือก p ที่เป็นจำนวนเฉพาะ

ข้อมูลใด ๆ ก็เปลี่ยนเป็นจำนวนเต็มได้

- double หรือ float → จำนวนเต็ม
  - Double.doubleToLongBits(d)
  - Float.floatToIntBits(f)
- boolean : true → 1, false → 0
- สตริง → จำนวนเต็ม
  - ข้อมูลเป็นสตริงภาษาอังกฤษตัวใหญ่ ก็มองเป็นเลขฐาน 26
    "DATA" → $3x26^3 + 0x26^2 + 19x26^1 + 0x26^0 = 53222$
- อ็อบเจกต์ → จำนวนเต็ม
  - แปลงข้อมูลภายในให้เป็นจำนวนเต็มแล้วนำมา "รวม" กัน

ตัวอย่าง

```
public int h(String s) {
  int hash = 0;
  for (int i=0; i<s.length(); i++)
    hash = 31 * hash + s.charAt(i);
  return (hash & 0x7FFFFFFF);
}
```

```
public int h(Point2D p) {
  long bits = Double.doubleToLongBits(p.getX());
  bits ^= Double.doubleToLongBits(p.getY()) * 31;
  hash = (((int) bits) ^ ((int) (bits >> 32)));
  return (hash & 0x7FFFFFFF);
}
```


การแฮชเอกภพ (Universal Hashing)

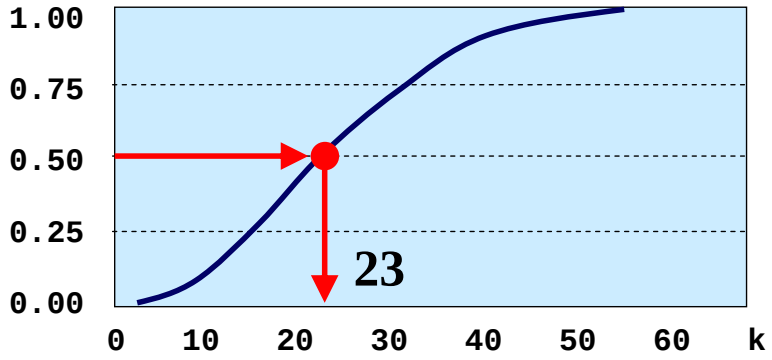
- วิธี hash ที่ผ่านมา เดาพฤติกรรมได้
  - ชุดข้อมูลที่ชนกันมากวันนี้ ก็จะชนกันมากตลอดไป
- ใช้สูตร $h(x) = ((ax + b) \% p) \% m)$
  - $x \in \{0, 1, ..., u - 1\}$, u คือจำนวนคีย์ที่เป็นไปได้
  - m คือขนาดตาราง
  - หา p ซึ่งคือจำนวนเฉพาะหนึ่งตัวในช่วง $[u, 2u)$
  - $0 < a < p$ และ $0 \le b < p$
- สุ่มเลือกค่า a และ b ก่อนใช้งาน
  - ชุดข้อมูลที่ชนกันมากวันนี้ อาจชนกันน้อยวันหน้า
  - สามารถพิสูจน์ได้ว่า จำนวนการชนเฉลี่ยเท่ากับ λ

ปฏิทรรศน์วันเกิด (Birthday Paradox)

- ต้องมีคนในห้องกี่คนขึ้นไป จึงจะโอกาสเกินครึ่งที่จะมีคนเกิดวันเดือนเดียวกันสองคนขึ้นไป

k คน โอกาสที่มีวันเกิดไม่ซ้ำกัน = $\left(\frac{366}{366}\right)\left(\frac{365}{366}\right)\left(\frac{364}{366}\right)\cdots\left(\frac{366-k+1}{366}\right)$

$$1-\left(\left(\frac{366}{366}\right)\left(\frac{365}{366}\right)\left(\frac{364}{366}\right)\cdots\left(\frac{366-k+1}{366}\right)\right)>0.5$$



ฟังก์ชันแฮชในจาวา

- คลาส Object มีเมท็อดชื่อ hashCode() โดยที่
  - ถ้า x.equals(y) เป็นจริง, x.hashCode() ต้อง == y.hashCode()
- hashCode ที่คลาส Object คืนค่าตำแหน่งเริ่มต้นของออปเจกต์ในหน่วยความจำ
  - ออปเจกต์ต่างกัน ได้ hashCode ต่างกัน
  - value object ควร overrides hashCode ให้ออปเจกต์สองตัวที่มีค่าเท่ากันต้องมี hashCode เหมือนกัน

```
System.out.println(new Integer(1234).hashCode());
System.out.println(new Integer(1234).hashCode());
System.out.println(new Object().hashCode());
System.out.println(new Object().hashCode());
```

```
1234
1234
8222510
18581223
```

ตัวอย่างการเขียน hashCode()

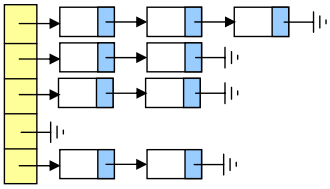
```
public class Point2D {
  private double x, y;
  ...
  public int hashCode() {
    long bits = Double.doubleToLongBits(x);
    bits ^= Double.doubleToLongBits(y) * 31;
    return (((int) bits) ^ ((int) (bits >> 32)));
  }
}
```

```
public class Book {
  private String name;
  private String publisher;
  private double price;
  ...
  public int hashCode() {
    return name.hashCode() ^ publisher.hashCode();
  }
}
```

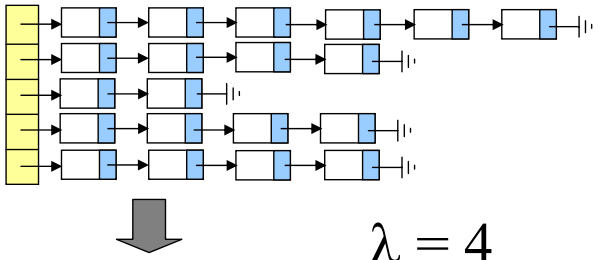

อีกครั้ง : SeparateChaining

```
public class SeparateChaining {
  ...
  public boolean contains(Object e) {
    return table[h(e)].contains(e);
  }
  public void add(Object e) {
    table[h(e)].add(0, e);
    ++size;
  }
  public void remove(Object e) {
    int i = h(e);
    int s = table[i].size();
    table[i].remove(e);
    if (s > table[i].size()) size--;
  }
  private int h(Object x) {
    return Math.abs(x.hashCode()) % table.length;
  }
}
```

Rehashing

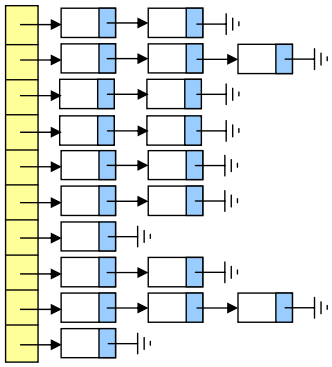


$\lambda = 1.8$



$\lambda = 4$

Rehashing



$\lambda = 2$

ถ้าฟังก์ชันแฮชกระจายดี

การลบและค้นใช้เวลา $O(\lambda)$

ถ้าควบคุม λ ไม่ให้เกินค่าคงตัว k

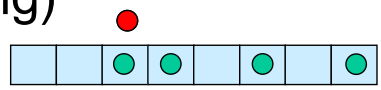
การลบและค้นใช้เวลาคงตัว

SeparateChaining : Rehash

```
public void add(Object e) {
  table[h(e)].add(0, e);
  ++size;
  if (size/table.length >= threshold) rehash();
}
private void rehash() {
  LinkedList[] oldTable = table;
  table = new LinkedList[2*table.length];
  for (int i=0; i<table.length; i++)
    table[i] = new LinkedList();
  for (int i=0; i<oldTable.length; i++) {
    Object[] items = oldTable[i].toArray();
    for (int j=0; j<items.length; j++) {
      table[ h(items[j]) ].add(0, items[j]);
    }
  }
}
```

การแก้ปัญหาการชนแบบอื่น

- แบบแยกกันโยง (separate chaining)
  - แต่ละช่องในตารางเก็บรายการโยงของข้อมูล
  - ข้อมูลที่ชนกันเก็บอยู่ด้วยกัน ไม่กระทบข้อมูลอื่น
  - เปลืองตัวโยง (links)
- แบบเลขที่อยู่เปิด (open addressing)
  - แต่ละช่องในตารางเก็บข้อมูล



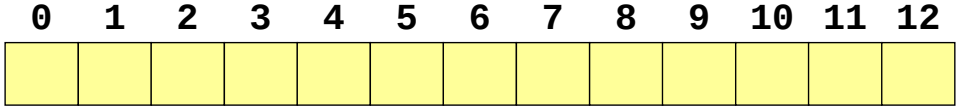
  - ถ้าชน ก็หาช่องว่างใหม่ในตารางเพื่อเก็บข้อมูล
  - $\lambda = n/m \leq 1$ เสมอ ต้องคุมไม่ให้เกินเกณฑ์ ($\lambda \leq 0.5$)
  - มีหลายวิธีในการหาช่องว่างใหม่ในตาราง เมื่อเกิดการชน
    - การตรวจเชิงเส้น (linear probing)
    - การตรวจกำลังสอง (quadratic probing)
    - การตรวจสองชั้น (double hashing)

การตรวจเชิงเส้น (Linear Probing)

- เมื่อชน หาช่องว่างถัดไปด้วยวิธีดูตัวถัดไปเรื่อย ๆ
- ให้ $h_j(x)$ คือช่องที่ probe หลังจากชนครั้งที่ j
- $h_0(x) = h(x)$ คือช่องที่ hash เริ่มต้น (home address)

$$h_j(x) = (h(x) + j) \% m$$

$$h_j(x) = (h_{j-1}(x) + 1) \% m$$



ใช้ `h(x) = x % 13` แล้วเพิ่มข้อมูลที่มีคีย์ตามลำดับดังนี้

`17 32 26 7 4 43 12 11 24`

LinearProbingHashSet

```
public class LinearProbingHashSet implements Set {
  private Object[] table;
  private int size = 0;
  public LinearProbingHashSet(int m) {
    table = new Object[m];
  }
  public boolean contains(Object e) {
    return table[indexOf(e)] != null;
  }
  private int indexOf(Object e) {
    int h = h(e);
    for (int j=0; j<table.length; j++) {
      if (table[h] == null) return h;
      if (table[h].equals(e)) return h;
      h = (h + 1) % table.length;
    }
    throw new AssertionError("ตารางเต็มได้ไง");
  }
  private int h(Object e) {
    return Math.abs(e.hashCode()) % table.length;
  }
```


LinearProbingHashSet

```
  public void add(Object e) {
    int i = indexOf(e);
    if (table[i] == null) {
      table[i] = e;
      ++size;
    }
  }
  public void remove(Object e) {
    int i = indexOf(e);
    if (table[i] != null) {
      table[i] = null;
      --size;
    }
  }
```

ผิด

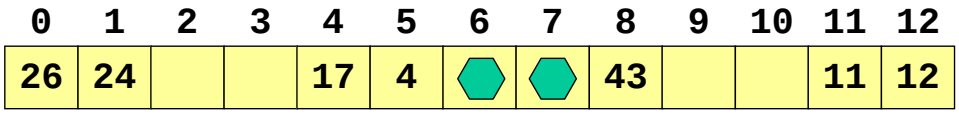
0	1	2	3	4	5	6	7	8	9	10	11	12
26	24			17	4	32	7	43			11	12

43

h(x) = x % 13

สถานะของช่องเก็บข้อมูล

- แต่ละช่องมี 3 สถานะ
  - ช่องว่าง ๆ ไม่เคยมีข้อมูลมาเก็บเลย `table[i] == null`
  - ช่องที่เก็บข้อมูลที่ถูกลบไปแล้ว `table[i] == DELETED`
  - ช่องที่มีข้อมูลเก็บอยู่ `table[i] != null && != DELETED`




```
public void remove(Object e) {
  int i = indexOf(e);
  if (table[i] != null) {
    table[i] = DELETED;
    --size;
  }
}
```

DELETED เป็นอ็อบเจกต์ที่ไม่เท่ากับอ็อบเจกต์อื่น การทำงานใน indexOf จะค้นต่อไปเมื่อพบ **DELETED**

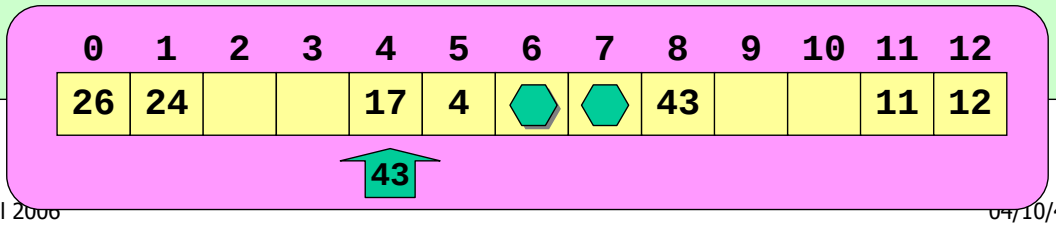
Rehash

```
public class LinearProbingHashSet implements Set {
  private static final Object DELETED = new Object();
  private Object[] table;
  private int size = 0;
  private int numNonNulls = 0;

  public void add(Object e) {
    int i = indexOf(e);
    if (table[i] == null) {
      table[i] = e;
      ++size; ++numNonNulls;
      if (numNonNulls > table.length/2) rehash();
    }
  }
  private void rehash() {
    Object[] old = table;
    table = new Object[4*size];
    size = numNonNulls = 0;
    for(int i = 0; i< old.length; i++)
      if (old[i] != null && old[i] != DELETED) add(old[i]);
  }
```

ไม่ได้ใช้ช่อง DELETED มาใช้ใหม่เลย

การนำช่อง DELETED มาใช้ใหม่

```
public void add(Object e) {
  int empty = -1;
  int h = h(e);
  for (int j=0; j<table.length; j++) {
    if (table[h] == DELETED && empty == -1) empty = h;
  int h = indexOf(e);                 ill || table[h].equals(e)) break;
    h = (h + 1) % table.length;
  }
  if (table[h] == null) {
    if (empty != -1) h = empty;
    table[h] = e;
    ++size; if (empty == -1) ++numNonNulls;
    if (numNonNulls > table.length/2) rehash();
  }
}
```



การเกาะกลุ่มปฐมภูมิ (Primary Clustering)

- ถ้าใช้ linear probling แล้วเพิ่มข้อมูลตัวใหม่อีกตัวลงตารางข้างล่างนี้ อยากทราบว่า ข้อมูลใหม่นี้จะถูกนำไปเก็บไว้ที่ช่องใดด้วยความน่าจะเป็นสูงสุด

การตรวจกำลังสอง (Quadratic Probing)

- เพื่อขจัดการเกาะกลุ่มปฐมภูมิ
- หลีกเลี่ยงการตรวจช่องติด ๆ กัน
- ให้ตรวจแบบก้าวกระโดดห่าง ๆ

+1, +3, +5, +7,...

$$h_j(x) = (h(x) + j^2) \% m$$

$$h_j(x) = (h_{j-1}(x) + 2j - 1) \% m$$

$$\begin{aligned} h_j(x) &= (h(x)+j^2) \% m \\ h_{j-1}(x) &= (h(x)+(j-1)^2) \% m \\ h_j(x) - h_{j-1}(x) &= (j^2 - (j-1)^2) \% m \\ &= (j^2 - j^2 + 2j - 1) \% m \\ h_j(x) &= (h_{j-1}(x) + 2j - 1) \% m \end{aligned}$$

การตรวจกำลังสองไม่ตรวจทุกช่อง

- ลองเพิ่ม 30 อีกตัว ($h(x) = x \% 13$)

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1		17	4	5		7	8				

```
h(x) = 4
(4+1²)%13 = 5
(4+2²)%13 = 8
(4+3²)%13 = 0
(4+4²)%13 = 7
(4+5²)%13 = 3
(4+6²)%13 = 1
```

```
(4+7²)%13 = 1
(4+8²)%13 = 3
(4+9²)%13 = 7
(4+10²)%13 = 0
(4+11²)%13 = 8
(4+12²)%13 = 5
(4+13²)%13 = 4
...
```

มีช่องว่างอาจหาไม่พบ

เมื่อตารางมีขนาดเป็นจำนวนเฉพาะ

- การตรวจกำลังสองจะดูอย่างน้อยครึ่งหนึ่งของตาราง
- ดังนั้น ถ้า load factor ≤ ½ ก็สบายใจได้ว่าจะหาช่องว่างพบเมื่อเพิ่มข้อมูล
- พิสูจน์ : ให้ $0 \le i < j \le \lfloor m/2 \rfloor$ ถ้าข้างบนไม่จริง ต้องมีการ probe ครั้งที่ i และ j ที่ดูช่องซ้ำกัน

$$\begin{aligned} h(x) + j^2 &\equiv h(x) + i^2 && \bmod m \\ j^2 &\equiv i^2 && \bmod m \\ (j^2 - i^2) &\equiv 0 && \bmod m \\ (j - i)(j + i) &\equiv 0 && \bmod m \end{aligned}$$

- เป็นไปไม่ได้ : $(j - i)$ ไม่เป็น 0, $(j+i)$ ก็ไม่เป็น m อีกทั้ง $(j - i)(j+i) \% m \neq 0$ เพราะทั้งสองพจน์ $< m$ และ m เป็นจำนวนเฉพาะ

QuadraticProbingHashSet

- เหมือน LinearProbingHashSet ต่างกันแค่เปลี่ยน

```
h = (h + 1) % table.length
```

เป็น

```
h = (h + 2*j-1) % table.length
```

- ต้อง rehash เมื่อ load factor เกินครึ่ง
- ขนาดของตารางเป็นจำนวนเฉพาะตลอด

construct และ rehash

```
import java.math.BigInteger;
public class QuadraticProbingHashSet implements Set {
  private static final Object DELETED = new Object();
  private Object[] table;
  private int size, numNonNulls;

  public QuadraticProbingHashSet(int m) {
    table = new Object[nextPrime(m)];
  }
  private int nextPrime(int n) {
    BigInteger bi = new BigInteger(Integer.toString(n));
    return bi.nextProbablePrime().intValue();
  }
  private void rehash() {
    Object[] old = table;
    table = new Object[nextPrime(4*size)];
    size = numNonNulls = 0;
    for(int i = 0; i< old.length; i++)
      if (old[i] != null && old[i] != DELETED) add(old[i]);
  }
  ...
```

เช่น n=5000 จะได้ 5003

indexOf

```
  public boolean isEmpty() { return size == 0; }
  public int size() { return size; }

  public boolean contains(Object e) {
    return table[indexOf(e)] != null;
  }
  private int indexOf(Object e) {
    int h = h(e);
    for (int j=0; j<table.length; j++) {
      if (table[h] == null) return h;
      if (table[h].equals(e)) return h;
      h = (h + 2*j - 1) % table.length;
    }
    throw new AssertionError("ตารางเต็มได้ไง");
  }
  private int h(Object e) {
    return Math.abs(e.hashCode()) % table.length;
  }
  ...
```

$$h_j(x) = (h_{j-1}(x) + 2j - 1) \% m$$

remove และ add

```
public void remove(Object e) {
  int i = indexOf(e);
  if (table[i] != null) {
    table[i] = DELETED; --size;
  }
}
public void add(Object e) {
  int empty = -1;
  int h = h(e);
  for (int j=0; j<table.length; j++) {
    if (table[h] == DELETED && empty == -1) empty = h;
    if (table[h] == null || table[h].equals(e)) break;
    h = (h + 2*j-1) % table.length;
  }
  if (table[h] == null) {
    if (empty != -1) h = empty;
    table[h] = e;
    ++size; if (empty == -1) ++numNonNulls;
    if (numNonNulls > table.length/2) rehash();
  }
}
```


การเกาะกลุ่ม

- การเกาะกลุ่มปฐมภูมิ (primary clustering)
  - เห็นได้ด้วยตา ข้อมูลอยู่ติด ๆ กัน
  - กลุ่มที่โต ยิ่งมีโอกาสโตขึ้น
  - การค้นจะช้าเหมือนการค้นแบบลำดับ
- การเกาะกลุ่มทุติยภูมิ (secondary clustering)
  - ข้อมูลที่มี $h(x)$ เดียวกัน จะตรวจช่องในตารางเหมือนกัน
  - ระยะกระโดดของการตรวจแปรตามหมายเลขครั้งที่ชน
  - $h_j(x) = (h(x) + j) \% m,\ \ h_j(x) = (h(x) + j^2) \% m$
  - แก้ปัญหานี้ได้ โดยให้ข้อมูลที่มี $h(x)$ เดียวกัน ไม่จำเป็นต้องมีระยะโดดของการตรวจเหมือนกัน
  - ให้ระยะกระโดดคำนวณจากค่าของข้อมูล

การแฮชสองชั้น (Double Hashing)

- ใช้ฟังก์ชันแฮชอีกตัวเพื่อคำนวณระยะกระโดด
- ทำให้ชุดข้อมูลที่แฮชไปที่ช่องเดียวกัน อาจมีระยะกระโดดต่างกัน

$h_j(x) = (h(x) + j \cdot g(x)) \% m$ $h_j(x) = (h_{j-1}(x) + g(x)) \% m$

- โดยที่ $g(x) \% m \neq 0$ (เพื่อไม่ให้ย่ำอยู่กับที่) เช่น
  - $g(x) = R - (x \% R)$ R เป็นจำนวนเฉพาะ และ $R < m$
- และ ตัวหารร่วมมากของ $g(x)$ และ m ต้องเป็น 1 จะได้ตรวจทุกช่องในตาราง
  - ประกันเงื่อนไขนี้ได้โดยให้ m เป็นจำนวนเฉพาะ
  - $h(x) = 0, g(x) = 4, m = 8$ จะตรวจช่อง 0 และ 4 เท่านั้น
  - $h(x) = 0, g(x) = 4, m = 7$ จะตรวจช่อง $0, 4, 1, 5, 2, 6, 3$

DoubleHashingHashSet

```
public class DoubleHashingHashSet implements Set {
  ...

  private int indexOf(Object e) {
    int h = h(e);
    int g = g(e);
    for (int j=0; j<table.length; j++) {
      if (table[h] == null) return h;
      if (table[h].equals(e)) return h;
      h = (h + g) % table.length;
    }
    throw new AssertionError("ตารางเต็มได้ไง");
  }
  private int h(Object e) {
    return Math.abs(e.hashCode()) % table.length;
  }
  private int g(Object e) {
    return 11 - (Math.abs(e.hashCode()) % 11);
  }
  ...
```

$h_j(x) = (h_{j-1}(x) + g(x)) \% m$

เปรียบเทียบการเกาะกลุ่ม

การตรวจเชิงเส้น (linear probing)

การตรวจกำลังสอง (quadratic probing)

การแฮชสองชั้น (double hashing)

$$\lambda = 0.8$$

เปรียบเทียบจำนวนการตรวจเฉลี่ย

- การตรวจเชิงเส้นตรวจจำนวนช่องมากกว่าแบบอื่น
- การตรวจกำลังสองและแบบสองชั้นใกล้เคียงกัน
- ถ้า $\lambda \leq 0.5$ ทั้งสามแบบไม่ต่างกันมาก

	Linear Probing			Quadratic Probing			Double Hashing	
	พบ	ไม่พบ		พบ	ไม่พบ		พบ	ไม่พบ
$\lambda = 0.3$	1.21	1.52		1.21	1.47		1.19	1.43
$\lambda = 0.4$	1.33	1.89		1.31	1.75		1.28	1.67
$\lambda = 0.5$	1.50	2.50		1.43	2.14		1.39	2.02
$\lambda = 0.6$	1.75	3.63		1.59	2.72		1.53	2.54
$\lambda = 0.7$	2.16	6.02		1.82	3.70		1.74	3.44
$\lambda = 0.8$	3.00	12.84		2.16	5.64		2.05	5.32
$\lambda = 0.9$	5.44	49.70		2.79	11.37		2.67	11.63

เปรียบเทียบจำนวนการตรวจเฉลี่ย

	จำนวนการตรวจเฉลี่ย	
	หาพบ	หาไม่พบ
แบบแยกกันโยง $(\lambda \geq 0)$	$1 + \lambda/2$	$1 + \lambda$
การตรวจเชิงเส้น $(0 \leq \lambda \leq 1)$	$\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$
การแฮชสองชั้น $(0 \leq \lambda \leq 1)$	$\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$	$\frac{1}{1-\lambda}$

ถาม : เก็บข้อมูลโดยใช้การตรวจเชิงเส้น ถ้าต้องการตรวจโดยเฉลี่ยไม่เกิน 5 ครั้ง ต้องควบคุมให้ตารางแฮชมี λ เป็นเท่าใด

ตอบ : $5 \geq \frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$ $9 \geq \frac{1}{(1-\lambda)^2}$ $1-\lambda \geq \sqrt{1/9}$ $\lambda \leq 2/3$

เปรียบเทียบเวลาการทำงาน

1117=1x3x3x3/2x3x3x3x3x3/2/2x3x3/2/2/2/2/2x3x3x3/2/2/2x3/2

สร้าง **Set** ด้วย	เวลาการทำงาน (ms)
ArraySet	**164987**
BSTSet	**1112**
AVLSet	**430**
LinearProbingHashSet	**1903**
QuadraticProbingHashSet	**390**
SeparateChainingHashSet	**350**

ตอนทำงานเสร็จ set มีข้อมูลจำนวน 73816 ตัว

เปรียบเทียบเนื้อที่

- แบบตรวจเชิงเส้น
  - ต้องการเก็บ 1200 ตัว
  - ให้ λ = 0.5, m = 2400
  - ตารางคืออาเรย์ของ object reference (ช่องละ 4 ไบต์)
  - ตารางใช้เนื้อที่ 4*2400 = 9600 ไบต์
  - ใช้เนื้อที่ทั้งหมด 9600 ไบต์
  - λ = 0.5, จากสูตรได้จำนวนการตรวจเฉลี่ยเป็น 2.5 ช่อง
- แบบแยกกันโยง
  - ต้องการตรวจเฉลี่ยจำนวน 2.5 ช่อง
  - จากสูตร ต้องให้ λ = 1.5
  - ต้องการเก็บ 1200 ตัว
  - ต้องมีตาราง 1200/1.5 = 800 ช่อง
  - ตารางใช้เนื้อที่ 4*800 = 3200 ไบต์
  - ถ้าใช้ singly linked list ไม่มีปมหัว ต้องมี 1200 ปม ใช้เนื้อที่ 1200*8 = 9600 ไบต์
  - รวมเป็น 3200+9600 = 12800 ไบต์

$$\#probes = \frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

$$\#probes = 1 + \lambda$$

แบบแยกกันโยงกับแบบเลขที่อยู่เปิด

- แบบแยกกันโยง
  - เปลืองตัวโยง
  - ไม่มีข้อจุกจิกเรื่องการลบข้อมูล
  - กลุ่มข้อมูลที่ชนกันเองมีผลกระทบในกลุ่มกันเอง ไม่มีผลต่อกลุ่มอื่น
- แบบกำหนดเลขที่อยู่เปิด
  - ประหยัดกว่า ถึงแม้จะมี λ ต่ำ
  - ข้อมูลอยู่ใกล้กัน ระบบ cache ทำให้เข้าถึงข้อมูลได้เร็วกว่าแบบโยงไปมา
  - การลบข้อมูลมีผลต่อการเกาะกลุ่มข้อมูล
  - ข้อมูลที่ชนกันจะมีผลกระทบกับข้อมูลอื่น ๆ

ข้อควรระวัง

- ไม่เหมาะกับบริการที่เกี่ยวข้องกับอันดับของข้อมูล
  - getMin, getMax, ...
  - ต้องค้นทั้งตาราง $\Theta(m+n)$
- ต้องระวังเรื่องฟังก์ชันแฮช

```
public class Book {
  private String isbn;
  ...
  public int hashCode() {
    return isbn.hashCode() | &7FFFFFFFH;
  }
}
```

สรุป

- การค้น เพิ่ม ลบข้อมูลในตารางแฮชทำได้รวดเร็ว
- สามารถปรับเวลาการทำงานให้เร็วขึ้น ด้วยการใช้เนื้อที่เข้าแลก เพื่อให้ได้ λ ที่เหมาะสม
- ฟังก์ชันแฮชมีผลต่อประสิทธิภาพการทำงาน