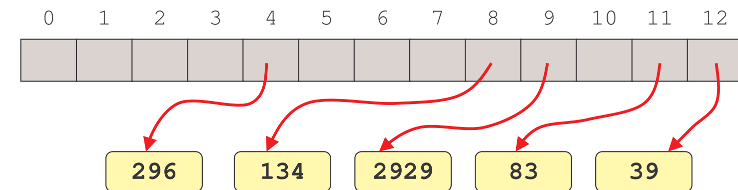
2110211 โครงสร้างข้อมูลเบื้องต้น

Hash Tables

สมชาย ประสิทธิ์จูตระกูล

การคำนวณตำแหน่งที่เก็บของข้อมูล

- มีข้อมูล n ตัว
- จองอาเรย์ขนาด m ช่อง $(m \geq n)$
- หาฟังก์ชัน f(x)
  - คำนวณ index ของอาเรย์ที่เก็บ x
  - $0 \leq f(x) < m$
  - ถ้า $x \neq y$ แล้ว $f(x) \neq f(y)$



`f(x) = (ผลรวมของเลขโดดทุกตัวใน x) % 13`


```
public class Table implements Set {
  private Object[] table;
  private int size = 0;
  public Table(int m) {
    table = new Object[m];
  }
  public void add(Object x) {
    if (table[f(x)] == null) ++size;
    table[e.hashCode()] = x;
  }
  public void remove(Object x) {
    if (table[f(x)] != null) --size;
    table[e.hashCode()] = null;
  }
  public void contains(Object x) {
    return table[f(x)] != null;
  }
  public void isEmpty() { return size == 0; }
  public int size() { return size; }

  private int f(Object x) { ... }
}
```

เวลาการทำงานขึ้นอยู่กับเวลาการคำนวณของ `f(x)`

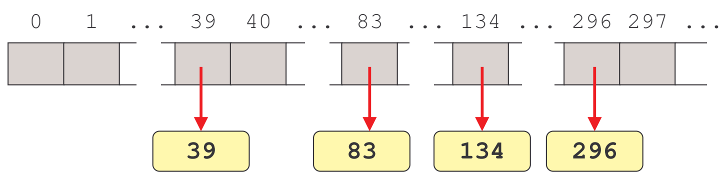
จากนี้ไปถือว่าข้อมูลเป็นเลขจำนวนเต็ม

- ถ้าไม่ใช่จำนวนเต็ม ก็สามารถแปลงเป็นจำนวนเต็มได้
  - มองออปเจกต์เป็น bit string ซึ่งก็มองเป็นจำนวนเต็มได้,
    - $11111010100111000010100_2 = 8211988_{10}$
    - ข้อมูลเป็น double หรือ float ก็เปลี่ยนเป็น long หรือ int Double.doubleToLongBits(d), Float.floatToIntBits(f)
  - มองเป็นเลขฐาน ??
    - ข้อมูลเป็นสตริงภาษาอังกฤษตัวใหญ่ ก็มองเป็นเลขฐาน 26 "DATA" -> $3 \times 26^3 + 0 \times 26^2 + 19 \times 26^1 + 0 \times 26^0 = 53222$
  - ข้อมูลหลายประเภทรวมกันก็มองเป็นเลขหลายตำแหน่ง
    - 123,true,DATA ก็ให้มีค่าเป็น 123153222
- ไม่จำเป็นต้องใช้ทุก fields ของออปเจกต์ แต่ขอให้ออปเจกต์ที่ต่างกัน แทนด้วยจำนวนเต็มต่างกัน
  - เช่น ใช้เฉพาะ ID ของข้อมูลนิสิตจุฬาฯ แทนตัวข้อมูล

เลือกเฉพาะส่วนคีย์ (key) ของข้อมูลมาแปลงเป็นจำนวนเต็ม

หา f(x) ได้อย่างไร ?

- Direct-addressing table : f(x) = x
  - ใช้ค่าของคีย์เป็น index ของอาเรย์
  - เร็ว, f(x) คำนวณง่าย add, remove, contains เป็น O(1)
  - เปลืองเนื้อที่มาก ๆ
    - ต้องจองอาเรย์ขนาดเท่ากับค่ามากสุดของคีย์ (+1)



ให้ f(x) ซับซ้อนขึ้น อาจลดขนาดของตารางได้

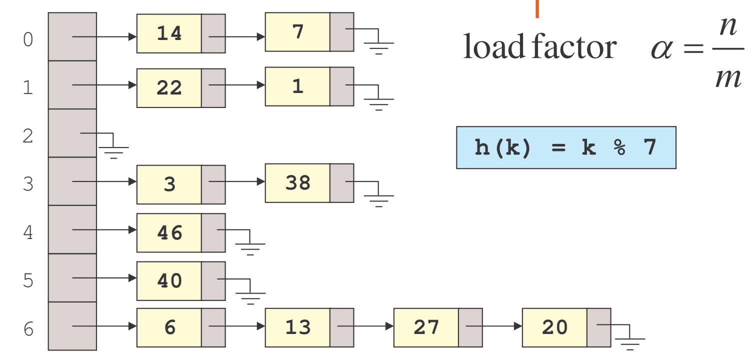
- ถ้าพิจารณาคีย์ให้ดี ๆ อาจตัดบางส่วนทิ้งได้ เช่น
  - ต้องการเก็บข้อมูลของนิสิต 8 คนนี้ โดยดูเฉพาะรหัสนิสิต
    - 4830102521, 4830630421, 4830805721, 4830556221, 4830601921, 4830683121, 4830795821, 4830901221
  - ถ้าใช้ f(x) = x ก็ต้องจองที่เก็บ 4830901222 ช่อง
  - ถ้าตัด 4830ซ้าย และ 21ขวา ออก ก็จองเหลือ 9013 ช่อง
  - ...
  - ... พยายามหา f(x) ที่ใช้อาเรย์ขนาดเล็กๆ
  - ...
  - ... หา f(x) ได้ไหม ที่ใช้อาเรย์น้อยสุดคือ 8 ช่อง !!!
- ถ้าอยากจะเพิ่มคีย์อีกตัว อาจต้องหา f(x) ใหม่ !

ทางออก

- ลดความทะเยอทะยาน
  - ไม่ต้องการ one-to-one function ($x \neq y \rightarrow f(x) = f(y)$)
  - ขอแบบ many-to-one ก็พอ
    - $x \neq y$ แต่ f(x) อาจ = f(y) ได้ เรียกว่า x "ชน" กับ y
- ด้วยวิธีนี้ทำให้
  - เกิดปัญหาการชน (คีย์หลายตัวแย่งกันใช้ช่องเดียวกัน)
  - แต่เราหาฟังก์ชันได้ง่ายขึ้น
  - จองอาเรย์ไม่ต้องใหญ่มโหฬาร
  - สามารถควบคุมเวลาของ add, remove, contains เป็น O(1) ได้โดยใช้เนื้อที่เข้าแลก
- เราจะศึกษา
  - hash function : h(k) ฟังก์ชันที่ทำให้เกิดการชนน้อย ๆ
  - collision resolution : วิธีจัดการปัญหาการชน

ดูวิธีจัดการปัญหาการชนแบบง่ายกันก่อน

- separate chaining
  - เก็บข้อมูลทั้งหลายที่ชนกันในช่องหนึ่ง ไว้เป็น list ที่ช่องนั้น
  - n คือจำนวนข้อมูล, m คือขนาดของตาราง
  - ถ้า h(k) กระจายคีย์ได้ดี แต่ละ list จะยาวเฉลี่ย n/m
  - การเพิ่ม O(1), การค้นและการลบ O(1 + n/m)



SeparateChainingHashSet

```
public class SeparateChainingHashSet implements Set {
  private LinkedList[] table;
  private int size = 0;

  public SeparateChainingHashSet(int cap) {
    table = createAndInitializeTable(cap);
  }
  private LinkedList[] createAndInitializeTable(int cap) {
    LinkedList[] table = new LinkedList[cap];
    for (int i = 0; i < table.length; i++)
      table[i] = new LinkedList();
    return table;
  }
  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }
  public boolean contains(Object element) {
    return table[h(element)].contains(element);
  }
  private int h(Object x) {  ...  }
  ...
```


SeparateChainingHashSet

```
public class SeparateChainingHashSet implements Set {
  ...

  public void add(Object element) {
    if (!contains(element)) addEntry(element);
  }
  private void addEntry(Object element) {
    table[h(element)].add(0, element);
    ++size;
  }
  public void remove(Object element) {
    int i = h(element);
    int s = table[i].size();
    table[i].remove(element);
    if (s > table[i].size()) size--;
  }
}
```

เป็นเซตต้องไม่ให้ซ้ำ ก็ไม่ใช่ O(1)

ถ้า load factor มากเกินไป → ช้า

- ก็ขยายตาราง แล้ว rehash ทำให้ load factor ต่ำลง

```
public void add(Object element) {
  if (!contains(element)) {
    putEntry(element);
    if (size/table.length > threshold) rehash();
  }
}
private void rehash() {
  LinkedList [] oldTable = table;
  table = createAndInitializeTable(2*oldTable.length);
  size = 0;
  for(int i = 0; i<oldTable.length; i++) {
    Object[] items = oldTable[i].toArray();
    for(int j=0; j<items.length; j++) {
      addEntry(items[j]);
    }
  }
}
```


กลับมาที่ Hash Functions

- ให้ U คือ universe ของคีย์ที่เป็นไปได้
  U = {0, 1, 2, ..., u – 1}
- ให้ m คือขนาดของ hash table
- h(k) มีหน้าที่แปลงค่า (พูดอีกอย่างว่า hash ค่า)
  - $\in$ U ไปเป็น j $\in$ {0, 1, ..., m – 1}
  - m < u แสดงว่ามีคีย์ชนกันแน่
- h(k) ที่ดีควร
  - คำนวณได้รวดเร็ว ใช้เวลาแปรตามขนาดของ x ไม่ใช่ u
  - เป็น simple uniform hashing
    - P(k) คือความน่าจะเป็นที่ x จะถูกเลือกจาก U
    - s_j = { x | h(x) = j }

$$\sum_{k \in s_j} P(k) = \frac{1}{m}$$

Simple Uniform Hashing

- ถ้า P(k) = 1/u
- นำ x ใน U ทุกตัวมา hash
- s_j ต้องเท่ากับ u/m
- ตีความได้ว่า
  - แต่ละช่องในตารางมีคีย์ที่ hash มาปริมาณเท่า ๆ กัน
  - h(k) จะ "โปรย" x ต่าง ๆ ให้กระจาย ๆ ในตาราง
  - เลือกคีย์มา 1 ตัว โอกาสที่ x จะถูก hash ลงช่องใดช่องหนึ่งใน m ช่องมีพอ ๆ กัน
  - เลือกคีย์มา n ตัวมา hash (n < m และ n << u) คีย์ทั้งหลายก็ย่อมกระจาย ๆ ในตาราง แสดงว่าชนกันน้อย

$$\sum_{k \in s_j} \frac{1}{u} = \frac{1}{m}$$

Simple Uniform Hashing

- ถ้าคีย์เป็นเลข 100 บิต U = $\{0,1,\ldots,2^{240} - 1\}$
- ตารางมีขนาด 2^{16} ช่อง
- นั่นหมายถึงต้องเปลี่ยนเลข 100 บิตเป็นเลข 16 บิต
- ถ้าคีย์ที่จะใช้ uniformly distributed ใน U
  - ให้ h(k) = k & 0xFFFF (หยิบเฉพาะ 16 บิตซ้ายของคีย์) ก็จะได้ simple uniform hashing และคำนวณได้เร็วมาก
- แต่ในทางปฏิบัติคีย์ที่ใช้จริงไม่กระจายอย่างที่คิด
  - P(k) ไม่เท่ากันหมด และก็ไม่รู้ P(k) ด้วย

Simple Uniform Hashing

- ข่าวร้าย
  - เรามักไม่รู้ P(k)
  - simple uniform hashing h(k) หายาก
- แต่มีกลวิธีง่าย ๆ ที่ใช้ได้ดีในทางปฏิบัติ
  - การวิเคราะห์เลขโดด (digit analysis)
  - การหาร (modulus hashing)
  - การคูณ (multiplicative hashing)
  - การพับ (folding)

การวิเคราะห์เลขโดด (Digit Analysis)

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

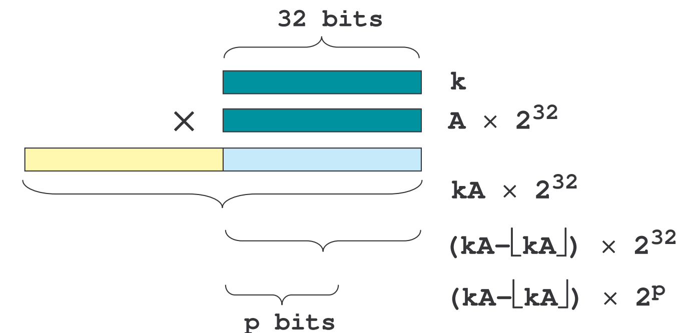
การหาร (Modulus Hashing) $h(k) = k \bmod p$

- หารคีย์ด้วยค่าคงตัว p แล้วเอาเศษเป็นผลลัพธ์
- เลือกค่า p ให้เหมาะสม ไม่ควรเลือก
  - $p = 2^q$ เพราะเลือกเฉพาะ q บิตซ้าย
  - $p = 10^q$ เพราะเลือกเฉพาะ q หลักซ้าย ถ้าคีย์เป็นฐานสิบ
  - $p = 2^q - 1$ และคีย์คือสตริงที่มองเป็นเลขฐาน 2^q เพราะการสลับตัวอักษรในสตริงจะได้ค่าเท่าเดิม
  - p ที่มีค่าน้อย ๆ หารลงตัว เพราะคีย์ที่มีค่าเป็นจำนวนเท่าของตัวหารนั้นซึ่งมีจำนวนมาก จะไม่กระจาย
- โดยทั่วไปเลือก p ที่มีค่าห่างจาก 2^q หรือจะให้ดีก็เลือก p เป็นจำนวนเฉพาะ

การคูณ (Multiplicative Hashing)

- คูณคีย์ด้วยจำนวนจริง z ที่มีค่าระหว่าง (0,1)
- แล้วคูณกับขนาดของตาราง ($m = 2^p$)

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$



การคูณ : Fibonacci Hashing

- ให้ A = golden ratio 0.6180339... จะช่วยแยกคีย์ที่มีค่าใกล้กันออกจากกัน $\hat{\phi} = \frac{\sqrt{5}-1}{2}$

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

```
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" : นำมา บวก, xor, ...

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

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

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

Additive Hashing

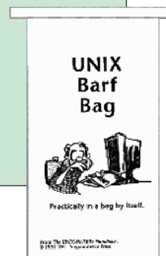
```
int additiveHash(String str) {
  int hash = 0;
  for(int i = 0; i < str.length(); i++) {
    hash += str.charAt(i);
  }
  return hash;
}
```


Rotating Hashing

```
int rotatingHash(String str) {
  int hash = str.length();
  for(int i = 0; i < str.length(); i++) {
    hash = ((hash << 5)^(hash >> 27))^str.charAt(i);
  }
  return (hash & 0x7FFFFFFF);
}
```


Additive + Rotating Hashing

```
long ELFHash(String str) {
  long hash = 0;
  long x    = 0;
  for(int i = 0; i < str.length(); i++) {
    hash = (hash << 4) + str.charAt(i);
    if((x = hash & 0xF0000000L) != 0) {
      hash ^= (x >> 24);
      hash &= ~x;
    }
  }
  return (hash & 0x7FFFFFFF);
}
```



ใช้แพร่หลายในระบบ UNIX

Universal Hashing

- วิธี hash ที่ผ่านมา เดาพฤติกรรมได้
  - ชุดข้อมูลที่ชนกันมากวันนี้ ก็จะชนกันมากวันหน้า
- Universal hashing
  $h(k) = ((ak + b) \% p) \% m)$
  - $k \in \{0, 1, ..., u-1\}$, m คือขนาดตาราง
  - p คือจำนวนเฉพาะหนึ่งในช่วง $[u, 2u]$
  - $0 < a < p$ และ $0 \le b < p$
- จะสุ่มเลือกค่า a และ b ไว้ก่อนใช้งาน
  - เดาพฤติกรรมไม่ได้
  - ชุดข้อมูลที่ชนกันมากวันนี้ อาจชนกันน้อยวันหน้า
  - สามารถพิสูจน์ได้ว่า expected number of collisions = load factor

รู้หรือยังว่าทำไมเรียก "Hash" Function

- www.webster.com
  - hash : to chop (as meat and potatoes) into small pieces

Hash Brown Potatoes

ปฏิทรรศน์วันเกิด (Birthday Paradox)

- ให้คนเป็นข้อมูล
- วัน-เดือนเกิด แทนผลของการแฮชคน (1 ถึง 366)
- คำถาม
  - ต้องมีคนในห้องกี่คนขึ้นไป จึงจะโอกาสเกินครึ่งที่จะมีคนเกิดวันเดือนเดียวกันสองคนขึ้นไป

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

การบ้าน : k มีค่าเท่าไร ?   (บอกคร่าว ๆ ก่อนก็ได้ว่า $k < 30$)

- แม้ฟังก์ชันแฮชที่ใช้จะดี และข้อมูลมีไม่มาก ก็มีโอกาสที่จะเกิดการชน

Hash Functions ในจาวา

- คลาส Object มีเมท็อดชื่อ hashCode() โดยที่
  - สำหรับออปเจกต์ x และ y 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("DATA".hashCode());
System.out.println(new Integer(1234).hashCode());
Sy                tln(new Integer(1234).hashCode());
Sy                tln(new Object().hashCode());
Sy                tln(new Object().hashCode());
```

```
2090922
1234
1234
8222510
18581223
```

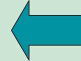

hashCode : ใช้เฉพาะ fields ที่ใช้ใน equals

```
public class Student {
  long id;
  String name;
  ...
  public boolean equals(Object o) {
    if (!(o instanceof Student)) return false;
    return id == ((Student)o).id;
  }
  public int hashCode() {
    int key = (int) (id ^ (id >>> 32));
    key += ~(key << 15);
    key ^=  (key >>> 10);
    key +=  (key << 3);
    key ^=  (key >>> 6);
    key += ~(key << 11);
    key ^=  (key >>> 16);
    return key & 0x7FFFFFFF;
  }
  ...
```

อีกครั้ง : SeparateChainingHashSet

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

hashCode อาจคืนจำนวนลบได้



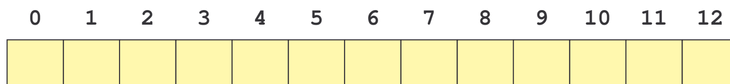
Collision Resolutions

- Separate chaining
  - ง่าย คุมเวลาได้ด้วยการขยายตารางเพื่อปรับ load factor
  - เปลือง links
- Open addressing
  - เก็บข้อมูลทั้งหมดในตารางแฮช
  - ถ้าชน ก็หาช่องว่างใหม่ในตารางแฮชเพื่อเก็บข้อมูล
  - n คือจำนวนข้อมูล m คือขนาดของตาราง $n \le m$
  - load factor = $n/m \le 1$ เสมอ ต้องคุมไม่ให้เกินเกณฑ์
  - มีหลายวิธีในการหาช่องว่างใหม่ในตาราง เมื่อเกิดการชน
    - linear probing
    - quadratic probing
    - double hashing

Linear Probing

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

$$h_j(k) = \left(h(k) + j\right) \ \% \ m$$



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

17, 32, 26, 7, 4, 43, 12, 11, 24

LinearProbingHashSet : แนวการทำงาน

```
public class LinearProbingHashSet implements Set {
  private Object[] table;
  private int size = 0;
  public LinearProbingHashSet(int cap) {
    table = new Object[cap];
  }
  public boolean contains(Object element) {
    return table[indexOf(element)] != null;
  }
  private int indexOf(Object element) {
    int h = h(element);
    int i = h;
    for(int j = 0; j<table.length; j++) {
      i = (h + j) % table.length;
      if (table[i] == null) break;
      if (table[i].equals(element)) break;
    }
    return i;
  }
  ...
```

ยังไม่สมบูรณ์

LinearProbingHashSet : แนวการทำงาน

```
public class LinearProbingHashSet implements Set {
  private Object[] table;
  private int size = 0;
  ...
  public void add(Object element) {
    int i = indexOf(element);
    if (table[i] == null) {
      table[i] = element;
      ++size;
    }
  }
  public void remove(Object element) {
    int i = indexOf(element);
    if (table[i] != null) {
      table[i] = null;
      --size;
    }
  }
  ...
}
```

ยังไม่สมบูรณ์

© S. Prasitjutrakul 2005

ลบโดยให้ table[i] = null ไม่ได้

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

`17, 32, 26, 7, 4, 43, 12, 11, 24`

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

จากนั้น `remove(4)` แล้วลองถาม `contains(43)` จะตอบว่าอะไร ?

© S. Prasitjutrakul 2005

ต้องระวังเรื่องช่องที่ข้อมูลถูกลบ

- แต่ละช่องมี 3 สถานะ
  - ช่องว่าง ๆ ไม่เคยมีข้อมูลมาเก็บเลย table[i] == null
  - ช่องที่มีข้อมูลเก็บอยู่ table[i] != null
  - ช่องที่เคยมีข้อมูลมาเก็บ แต่ถูกลบไปแล้ว
- จะรู้ว่าเป็นสถานะที่สามได้อย่างไร ?
  - ทำง่าย ๆ โดยการใช้ออบเจกต์พิเศษเก็บในช่องที่ถูกลบ

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

sentinel object

เป็น `singleton` มีแค่ออปเจกต์เดียวใช้ร่วมกันทั้งระบบ

© S. Prasitjutrakul 2005

LinearProbingHashSet : remove

```
public class LinearProbingHashSet implements Set {
  private static Object deleted = new Object[];
  private Object[] table;
  private int size = 0;
  private int numOccupied = 0;

  public LinearProbingHashSet(int cap) {
    table = new Object[cap];
  }

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

`numOccupied` มีไว้นับจำนวนช่องที่ไม่ใช่ `null` จะไว้ใช้ในภายหลัง

© S. Prasitjutrakul 2005

LinearProbingHashSet : contains

```
...
public boolean contains(Object element) {
  return table[indexOf(element)] != null;
}
private int indexOf(Object element) {
  int h = h(element);
  int i = h;
  for(int j = 0; j<table.length; j++) {
    i = (h + j) % table.length;
    if ( table[i] == null ) break;
    if ( table[i].equals(element) ) break;
  }
  return i;
}
...
```

เหมือนเดิม

LinearProbingHashSet : add

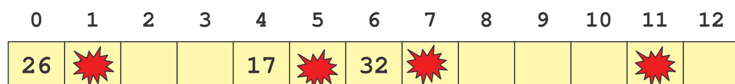
```
public void add(Object element) {
  int h = h(element);
  int i = h, empty = -1;
  for (int j = 0; j < table.length; j++) {
    i = (h + j) % table.length;
    if (table[i] == null || table == deleted) {
      if (empty == -1) empty = i;
      if (table[i] == null) break;
    }
    if (table[i].equals(element)) break;
  }
  if (table[i] == null) {
    if (table[empty] == null) ++numOccupied;
    table[empty] = element;
    ++size;
  }
}
```

วิ่งหาช่องว่างช่องแรกและวิ่งต่อไปดูว่าซ้ำกับข้อมูลเดิมหรือไม่

ขยายตารางเมื่อจำนวนช่อง non-null สูง

- จำนวนช่อง non-null สูง, ต้องวิ่งหาแบบลำดับ ช้า
- เรามี numOccupied เก็บจำนวน non-null ในตาราง
- โดยทั่วไป ควรขยายเมื่อ numOccupied เกินครึ่ง

```
public void add(Object element) {
  ...
  if (table[i] == null) {
    if (table[empty] == null) ++numOccupied;
    table[i] = element;
    ++size;
    if (numOccupied > table.length/2) rehash();
  }
}
```



Rehash

```
private void rehash() {
  Object [] oldT = table;

  table = new Object[2*table.length];
  size = numOccupied = 0;

  for(int i = 0; i< oldT.length; i++) {
    if (oldT[i] != null && oldT[i] != deleted) {
      add(oldT[i]);
    }
  }
}
```

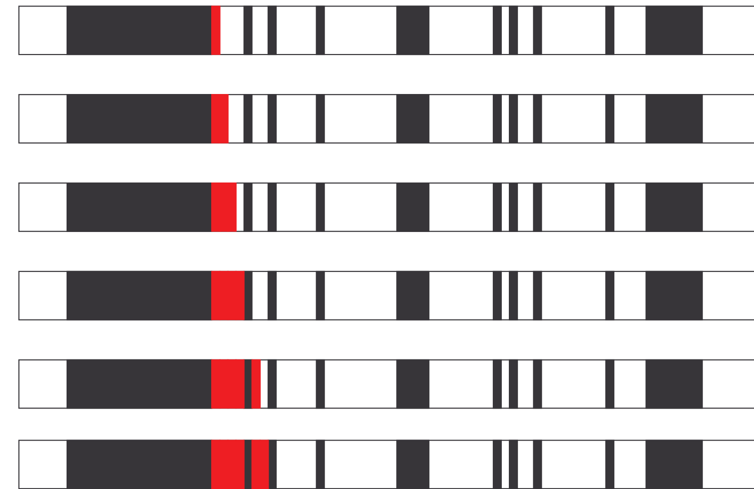
วิ่งทั้งตาราง

Primary Clustering

- ถ้าใช้ linear probling แล้วเพิ่มข้อมูลตัวใหม่อีกตัว ลงตารางข้างล่างนี้ อยากทราบว่า ข้อมูลใหม่นี้จะถูกนำไปเก็บไว้ที่ช่องใดด้วยความน่าจะเป็นสูงสุด

© S. Prasitjutrakul 2005

Cookie Monster Effect



© S. Prasitjutrakul 2005

Quadratic Probing

- เพื่อขจัดปัญหา primary clustering
- หลีกเลี่ยงการ probe ช่องถัดไป ๆ ๆ แบบ linear
- ให้ probe แบบก้าวกระโดด

$$h_j(k) = \left(h(k) + j^2\right) \% \ m$$

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

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

`4, 5, 8, 0, 7, 1,` **17**

© S. Prasitjutrakul 2005

Quadratic Probing : probe ทุกช่องหรือไม่ ?

- ลองเพิ่ม 30 อีกตัว

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

ใช้ `h(k) = k % 13`

ลำดับของ quadratic probe โดยเริ่ม home address ที่ 4 (ตารางขนาด 13) คือ

4,5,8,0,7,3,1,1,3,7,0,8,5,4,5,8,0,7,3,1,...

ลำดับของ quadratic probe โดยเริ่ม home address ที่ 4 (ตารางขนาด 16) คือ

4,5,8,13,4,13,8,5,4,5,8,13,4,13,8,5,4,5,8,13, ...

© S. Prasitjutrakul 2005

เมื่อตารางมีขนาดเป็นจำนวนเฉพาะ

- quadratic probing จะดูอย่างน้อยครึ่งหนึ่งของตาราง
- ดังนั้น ถ้า load factor ≤ ½ ก็สบายใจได้ว่าจะหาช่องว่างพบ เมื่อเพิ่มข้อมูล
- ให้ $0 \le i < j \le \lfloor m/2 \rfloor$ ถ้าข้างบนไม่จริง ต้องมีการ probe ครั้งที่ i และ j ที่ดูช่องซ้ำกัน

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

- เป็นไปไม่ได้ : (j – i) ไม่เป็น 0, (j+i) ก็ไม่เป็น m อีกทั้ง (j – i)(j+i) ก็ไม่มีทางเป็น m เพราะ m เป็น prime

QuadraticProbingHashSet : ขนาดตาราง

```
public class QuadraticProbingHashSet implements Set {
  private Object[] table;
  ...
  public QuadraticProbingHashSet(int cap) {
    table = new Object[nextPrime(cap)];
  }
  private void rehash() {
    Object [] oldT = table;
    table = new Object[nextPrime(2*table.length)];
    ...
  }
  private int nextPrime(int n) {
    BigInteger bi = new BigInteger(Integer.toString(n));
    return bi.nextProbablePrime().intValue();
  }
  ...
}
```


QuadraticProbingHashSet : สรุป

- เหมือน LinearProbingHash ต่างกันแค่เปลี่ยน

```
i = (h + j) % table.length
```

เป็น

```
i = (h + j*j) % table.length
```

- ต้อง rehash เมื่อ load factor เกินครึ่ง
- ขนาดของตารางเป็นจำนวนเฉพาะตลอด
- ก็จะปลอดจาก primary clustering
- แต่ก็ยังมีปัญหา secondary clustering
  - ชุดข้อมูลที่มี home address เดียวกัน จะมี probe sequence เหมือนกัน

Double Hashing

- ปัญหา secondary clustering มาจากการกำหนดสูตรการ "กระโดด" แบบตายตัว $h_j(k) = (h(k) + j^2) \% m$
- ควรเปลี่ยนระยะกระโดดให้แปรตามค่าของคีย์ด้วย
- ใช้อีก hash function หนึ่งเพื่อคำนวณระยะกระโดด
- ทำให้ชุดข้อมูลที่มี home address เดียวกัน อาจมีระยะกระโดดต่างกัน

$$h_j(k) = \left(h(k) + j \cdot g(k)\right) \;\%\; m, \quad g(k) > 0$$

เช่น

$g(k) = R - (k \% R)$ R เป็น prime และ R < m

ประสิทธิภาพของ Open Addressing

	ค้นพบ	ค้นไม่พบ
linear probing	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$
quadratic probing	?	?
double hashing	$\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$	$\frac{1}{1-\alpha}$

สรุปสารพัด data structures

List, Set, Map ที่สร้างด้วย	เมท็อดที่ใช้ในการค้น
array	equals
linked structure	equals
search tree	compareTo
hashing	equals, hashCode

สรุปสารพัด data structures

List, Set, Map ที่สร้างด้วย	เวลาการเพิ่ม ลบ ค้น
array list	มีบางเมท็อดเป็น O(n)
linked list	มีบางเมท็อดเป็น O(n)
balanced search tree	O(log n)
hash table	O(1)

อย่าลืมว่า hashing ทำให้คีย์ค่าใกล้ ๆ กัน hash ไปแล้วห่างกันแค่ไหนก็ไม่รู้
ดังนั้น hash table ทำงานช้ากับ operations ที่เกี่ยวข้องกับอันดับของข้อมูลเช่น
getMin, getMax, getNext, ...

คูณสองหารสามอีกครั้ง

```
public static void main(String[] args) {
  Set set = new ArraySet();
  Queue q = new ArrayQueue();
  Node v = new Node(1,null);
  q.enqueue(v); set.add(v);
  int target = 1117;
    while( !q.isEmpty() ) {
    v = (Node) q.dequeue();
    if (v.value == target) break;
    Node v1 = new Node(v.value*3, v);  // try x3
    Node v2 = new Node(v.value/2, v);  // try /2
    if (!set.contains(v1)) {q.enqueue(v1); set.add(v1);}
    if (!set.contains(v2)) {q.enqueue(v2); set.add(v2);}
  }
  if (v.value == target) printSolution(v);
}
```

ArraySet, BSTSet, AVLSet, LinearProbingHashSet, QuadraticProbingHashSet, SeparateChainingHashSet

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

สร้าง Set หลายได้แบบ

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

ตอนทำงานเสร็จ set มีข้อมูลจำนวน 73816 ตัว

© S. Prasitjutrakul 2005

การบ้าน : สร้าง map ด้วย hash table

- Map เป็นเสมือนที่เก็บคู่ลำดับ (key, value)
- ไม่มี key ซ้ำกันใน map
- การค้นจะให้ค่าของ key แล้วได้ value ที่คู่กัน คืนกลับมา
- จงสร้าง QuadraticProbingHashMap

```
public class QuadraticProbingHashMap implements Map
```

```
public interface Map {
  public int size();
  public boolean isEmpty();
  public boolean containsKey(Object key);
  public boolean containsValue(Object value);
  public Object get(Object key);
  public Object put(Object key, Object value);
  public void remove(Object key);
}
```

ข้อกำหนดของแต่ละเมท็อดหาอ่านได้จาก javadoc ไม่ต้องสร้างให้ครบเท่า java.util.Map เอาเท่าที่แสดงข้างบนนี้ก็พอ

© S. Prasitjutrakul 2005