Hash Table

ตารางแฮช

Outline

- ตารางเก็บข้อมูล
- Separate chaining hash table
- Open addressing hash table
  - Linear probing
  - Quadratic probing
  - Double hashing
- Hash functions

Map

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

ตารางเก็บข้อมูล

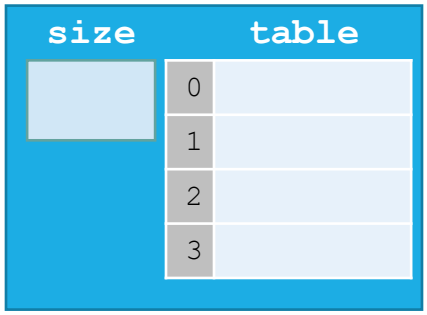
Table

AbstractTable

```
public abstract class AbstractTable implements Map {
  private Object[] table;
  private int size = 0;
  protected AbstractTable(int m) { ... }
  public boolean isEmpty()       { ... }
  public int size()              { ... }
  public boolean containsKey(Object key)       { ... }
  public Object get(Object key)                { ... }
  public Object put(Object key, Object value)  { ... }
  public void remove(Object x)                 { ... }
  protected abstract int f(Object key);   // เขียนใน subclass
}
```

AbstractTable

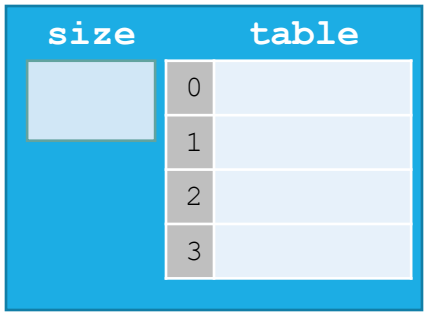
```
public abstract class AbstractTable implements Map {
  private Object[] table;
  private int size = 0;
```




```
  protected AbstractTable(int m) { table = new Object[m]; }
  public boolean isEmpty()       { return size == 0; }
  public int size()              { return size; }
  public boolean containsKey(Object key) {
    return table[f(key)] != null;
  }
  public Object get(Object key)  { return table[f(key)]; }
  public Object put(Object key, Object value) {...}
  public void remove(Object x) {...}
```

AbstractTable (cont.)

```
  public Object put(Object key, Object value) {
    Object oldValue = get(key);
    table[f(key)] = value;
    if (oldValue == null) ++size;
    return oldValue;
  }

  public void remove(Object x) {
    if (table[f(x)] != null) --size;
    table[f(x)] = null;
  }
  protected abstract int f(Object key); // เขียนใน subclass
}
```



การชน - Collision

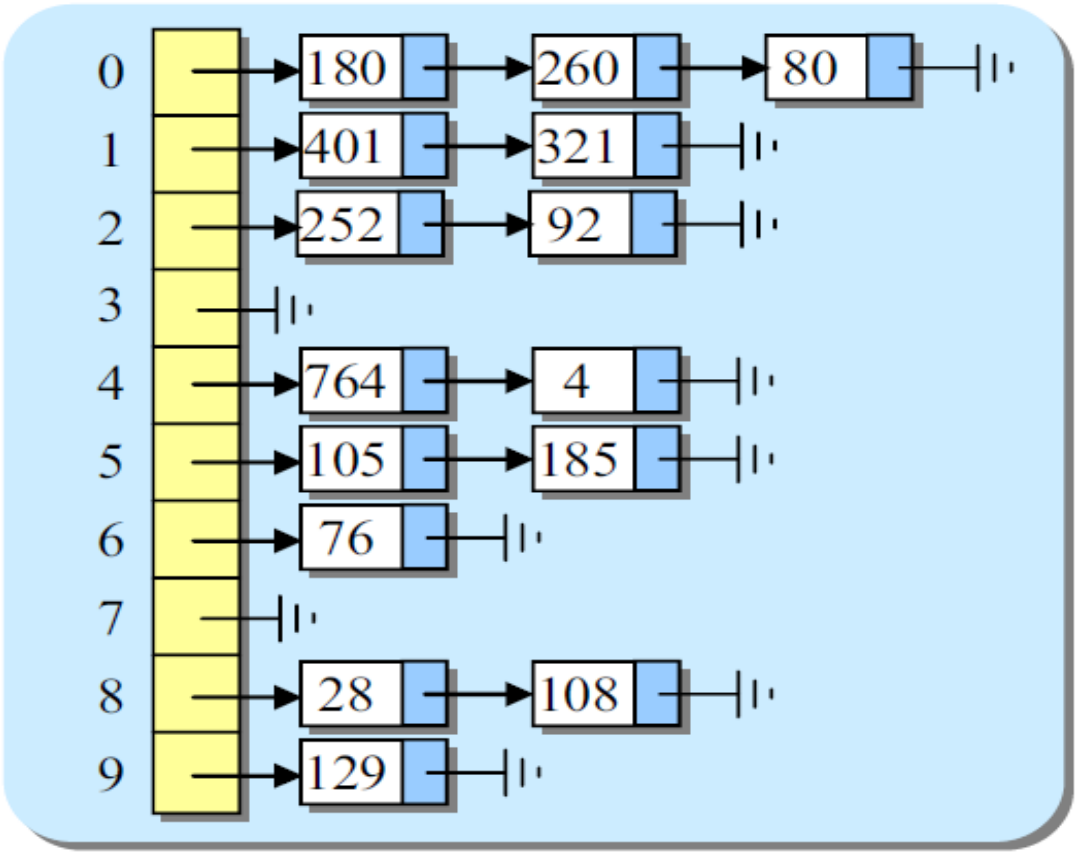
- ข้อมูลแต่ละตัวมี key ที่ไม่ซ้ำกับข้อมูลตัวอื่น
- ใช้ hash function คำนวณหาตำแหน่งที่จะเก็บข้อมูลจาก key ในข้อมูลนั้น
- Hash function อาจให้ค่าเดียวกันสำหรับ key 2 ตัว
- ถ้าตำแหน่งที่คำนวณมาจาก hash function มีข้อมูลที่มีค่า key อื่นเก็บอยู่ก่อนแล้ว เรียกว่า เกิดการชน

Separate Chaining Hash Table

Separate Chaining Hash Table

จัดการในกรณีที่เกิดการชน

- ค่าใน table แต่ละช่องเป็น **list** ของข้อมูลที่ประกอบด้วย key และ ค่าที่ผูกกับ key
- ข้อมูล**ทุกตัว**ที่มีค่าจาก hash function เท่ากัน จะเก็บอยู่ในช่องเดียวกันของ table โดยต่อกันเป็น list
- ถ้าเกิดการชน ให้เอาข้อมูลใส่ใน list ที่ตำแหน่งนั้น table



$h(x) = x\%10$

Example:

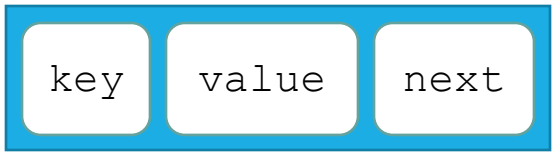
Separating Chaining Hash Table

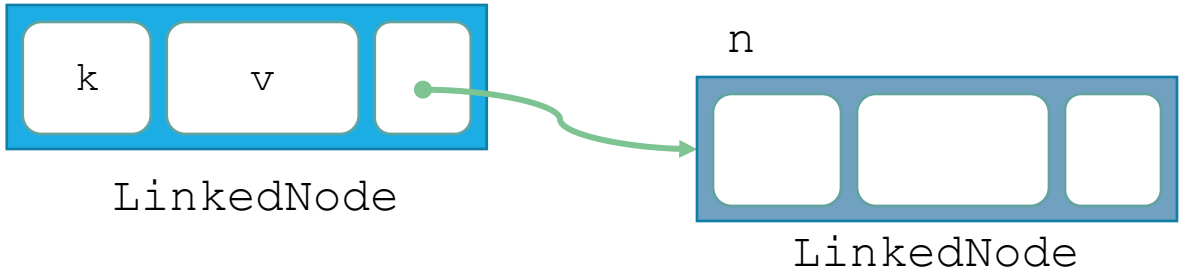
Separate Chaining Hash Table

```
public class SeparateChainingHashMap implements Map {
  private static class LinkedNode {...}
  private int size;
  private LinkedNode[] table;
  public SeparateChainingHashMap(int cap) {...}
  public int size() {...}
  public boolean isEmpty() {...}
  public Object get(Object key) {...}
  public boolean containsKey(Object key) {...}
  private LinkedNode getNode(Object key) {...}
  private int h(Object x) {...}
  public Object put(Object key, Object value) {...}
  public void remove(Object key) {...}
}
```

LinkedNode in SeparateChainingHashMap

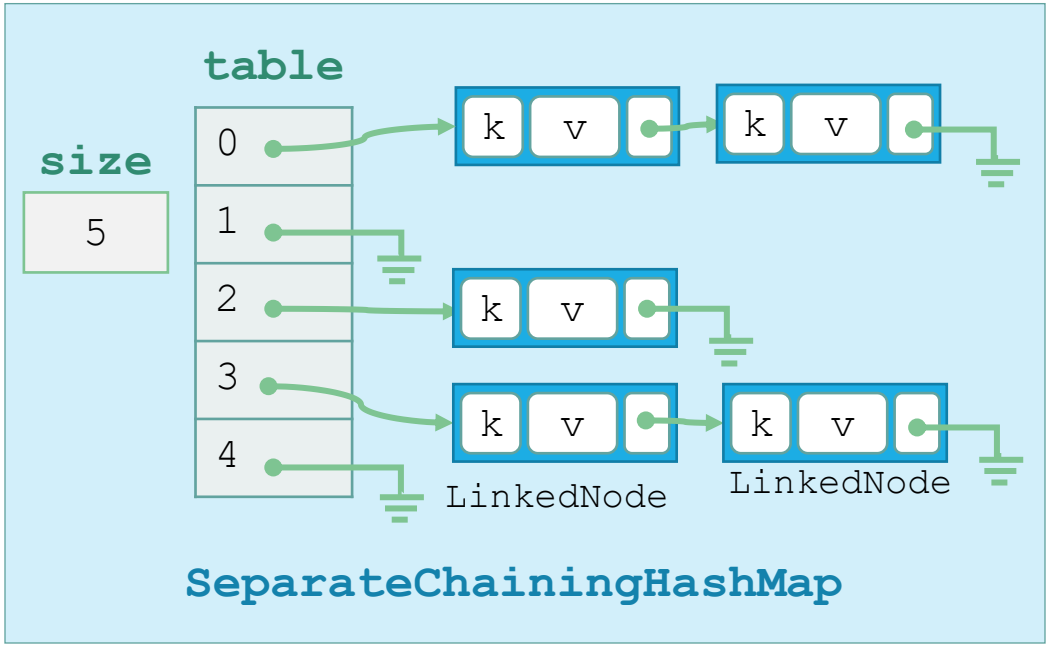
```
private static class LinkedNode {
  Object      key, value;
  LinkedNode next;

  LinkedNode(Object k, Object v, LinkedNode n) {
    key = k;    value = v;    next = n;
  }
}
```





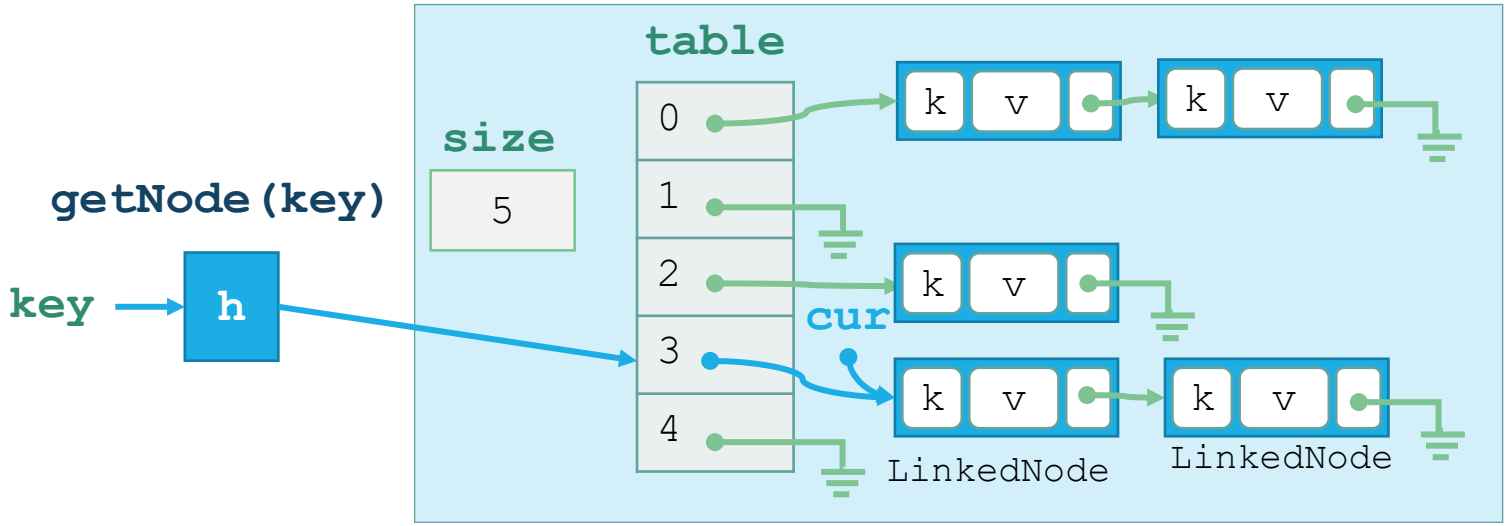
Attributes and Some Methods in SeparateChainingHashMap

```
public class SeparateChainingHashMap implements Map {
   ...
   private int size;
   private LinkedNode[] table;
   public SeparateChainingHashMap(int cap) {
      table = new LinkedNode[cap];
   }
   public int size() {
      return size;
   }
   public boolean isEmpty(){
      return size == 0;
   }
   ..
}
```



getNode in SeparateChainingHashMap

```
private LinkedNode getNode(Object key) {
  LinkedNode cur = table[h(key)];
  while (cur != null && !cur.key.equals(key)) cur = cur.next;
  return cur;       // return null if not found
}
```

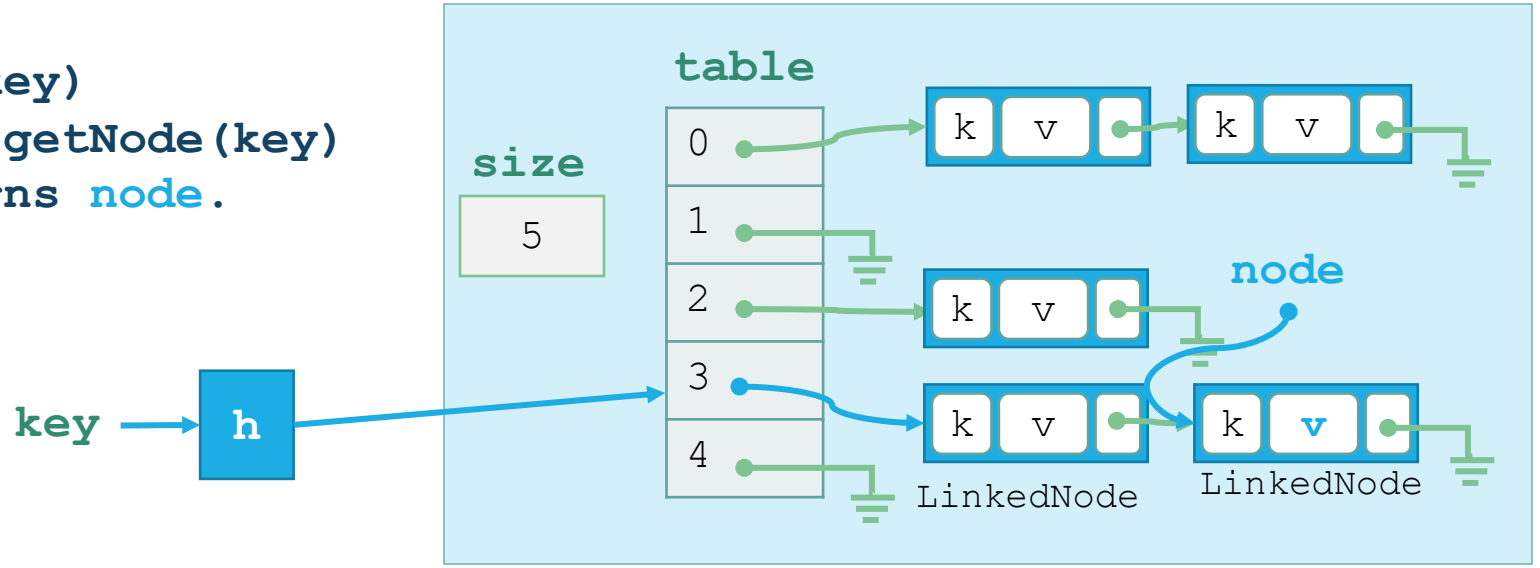



```
private int h(Object x) {
  return Math.abs(x.hashCode()) % table.length;
}
```

get and containsKey in SeparateChainingHashMap

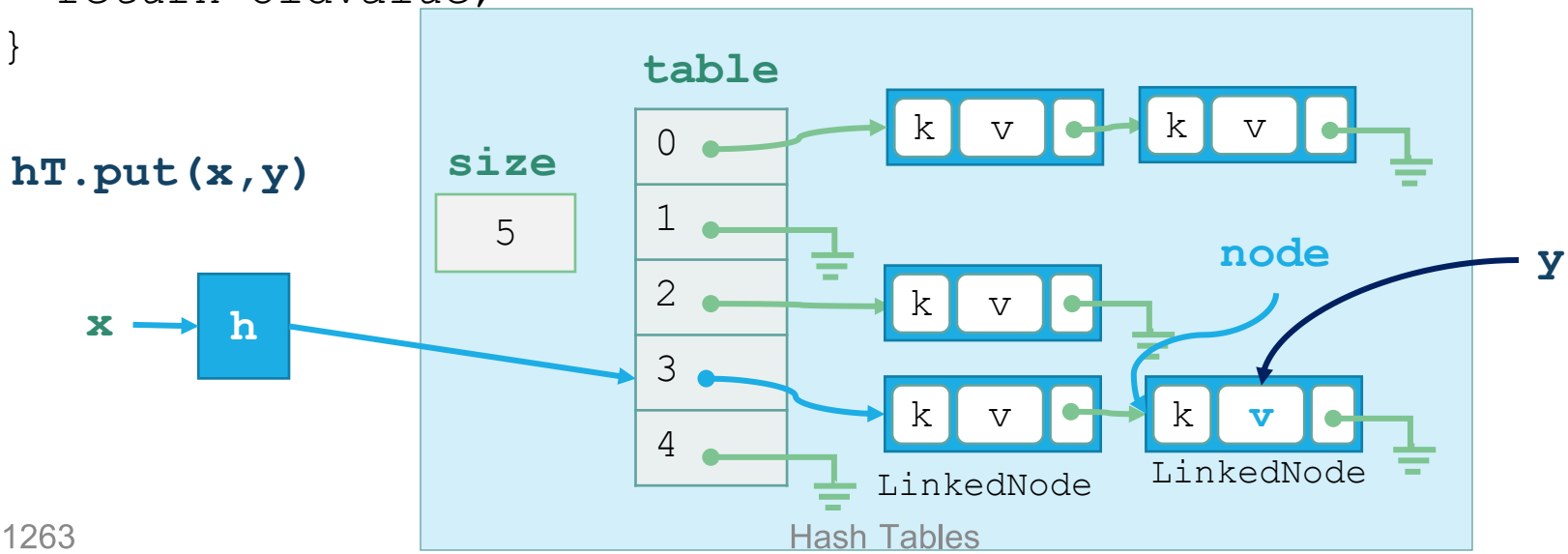
```
public Object get(Object key) {
  LinkedNode node = getNode(key);
  return node == null ? null : node.value;
}
public boolean containsKey(Object key) {
  return getNode(key)!=null;
}
```

get(key)
when getNode(key)
returns node.



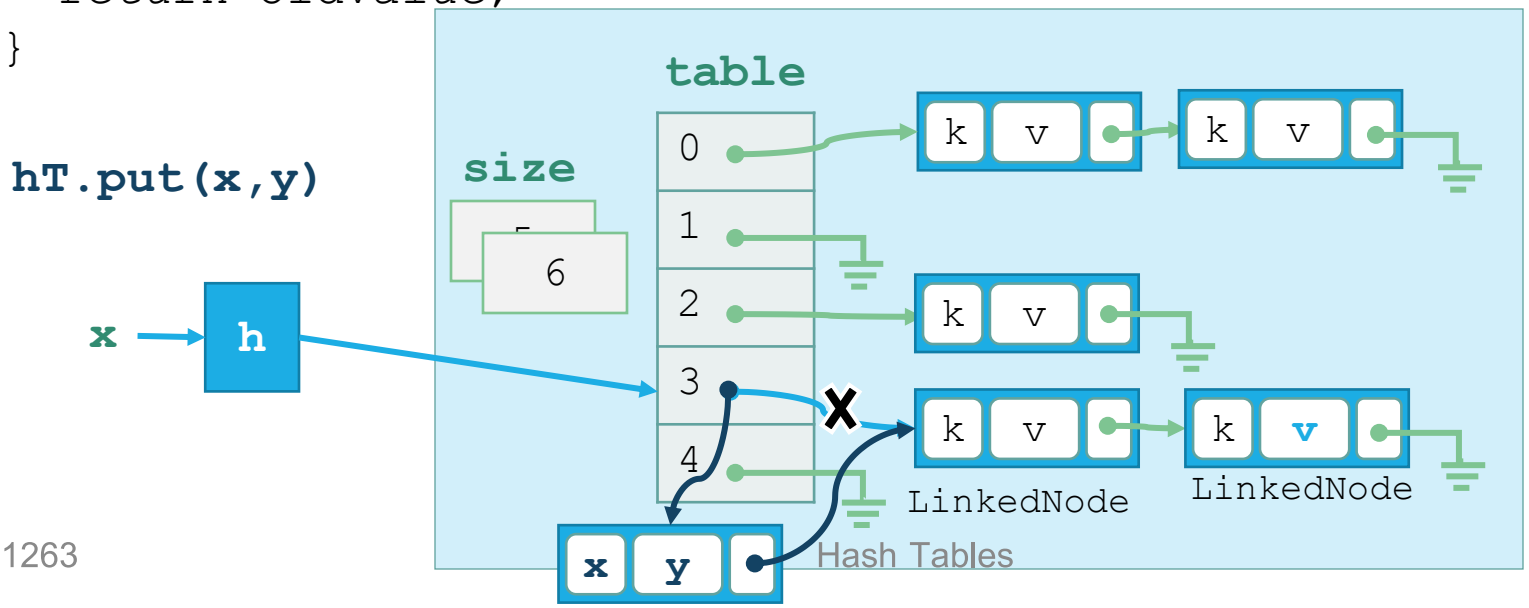
put in SeparateChainingHashMap

```
public Object put(Object key, Object value) {
  LinkedNode node = getNode(key);
  Object oldValue = null;
  if (node != null) {
    oldValue = node.value;  node.value = value;
  } else {
    int h = h(key);
    table[h] = new LinkedNode(key, value, table[h]);  ++size;
  }
  return oldValue;
}
```



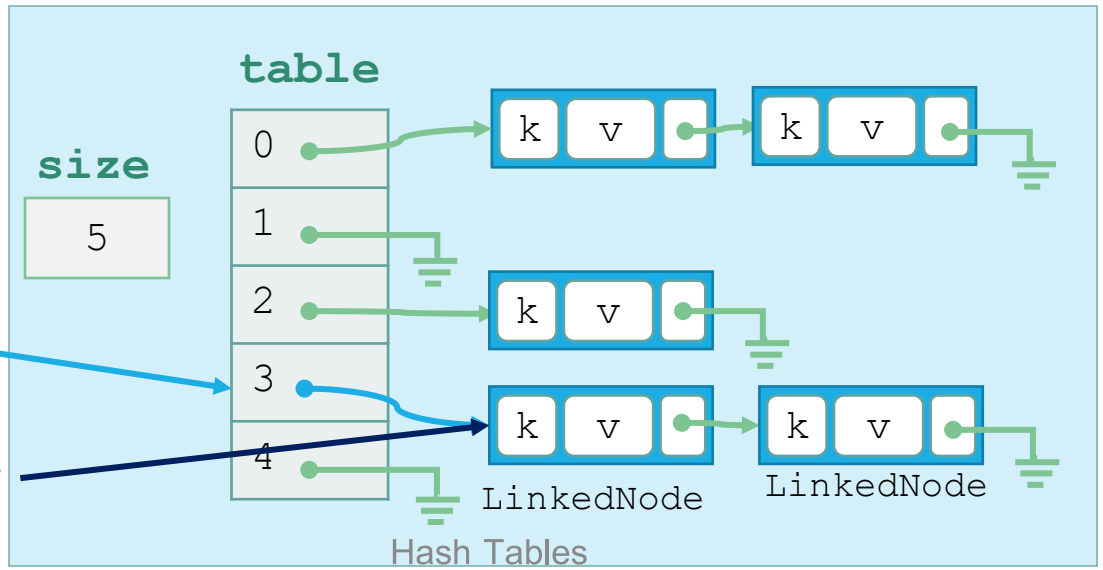
put in SeparateChainingHashMap

```
public Object put(Object key, Object value) {
  LinkedNode node = getNode(key);
  Object oldValue = null;
  if (node != null) {
    oldValue = node.value;  node.value = value;
  } else {
    int h = h(key);
    table[h] = new LinkedNode(key, value, table[h]);  ++size;
  }
  return oldValue;
}
```



remove in SeparateChainingHashMap

```
public void remove(Object key) {
   int h = h(key);
   if (table[h] == null) return;
   if (table[h].key.equals(key)){table[h]=table[h].next; --size;
   } else {
     LinkedNode prev = table[h];
     while (prev.next!=null && !prev.next.key.equals(key)) {
       prev = prev.next;
     }
     if (prev.next!=null) {prev.next = prev.next.next; --size;}
   }
}
}
```

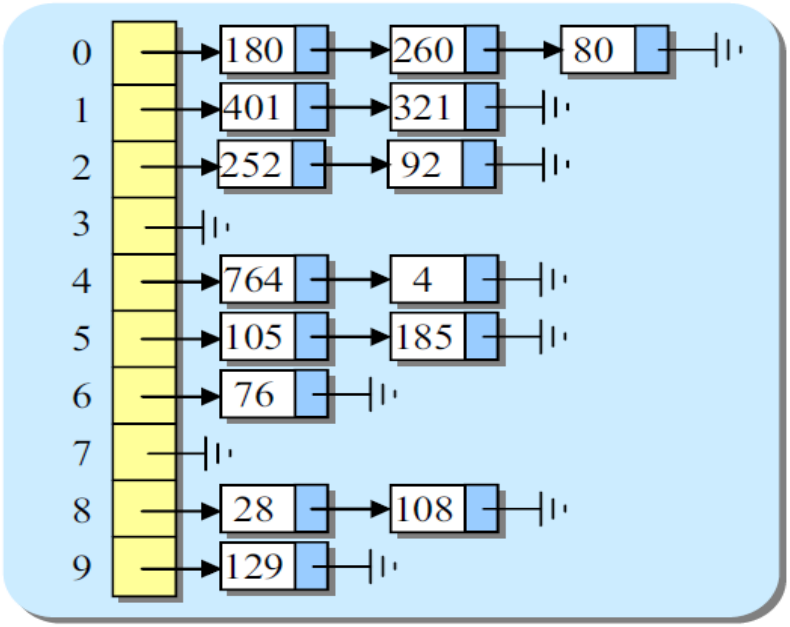


สัดส่วนบรรจุ - Load Factor λ

- Load factor = จำนวนข้อมูล / ขนาดของตาราง
- ถ้าตารางใหญ่และข้อมูลน้อย ความน่าจะเป็นในการชนจะต่ำ
- ถ้าการชนน้อย list แต่ละ list ในตารางจะไม่ยาว และการค้นหาค่าในตารางจะเร็ว

Load factor = 15/10 = 1.5

ความยาวของ List ในตารางนี้มีค่าตั้งแต่ 0 ถึง 3

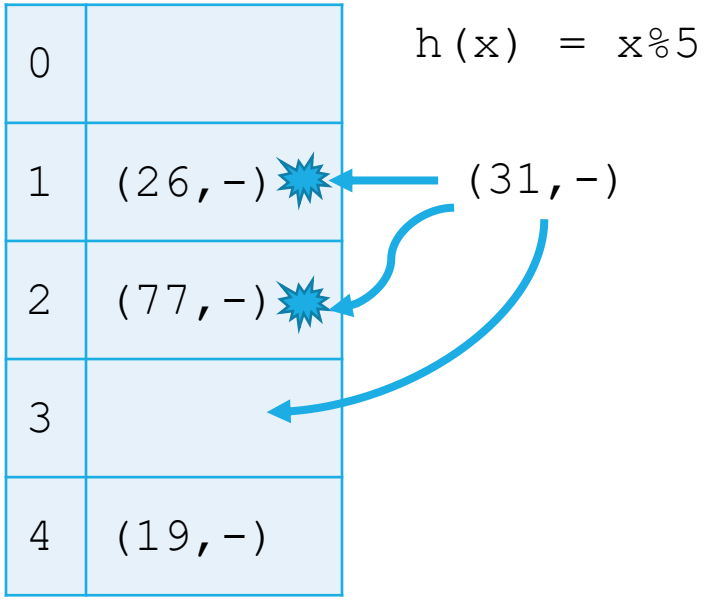


Open Addressing Hash Table

- Linear probing
- Quadratic probing
- Double hashing

Open Addressing Hash Table

- ไม่ใช้ chain
- ถ้าข้อมูลที่จะเก็บใน hash table ทำให้เกิดการชนกับข้อมูลที่เก็บไว้แล้ว ให้ตรวจ (probe) ที่ช่องอื่นต่อไป ถ้าชนอีก ก็ทำเหมือนเดิม จนไม่ชน
- ไม่ชน คือ เจอช่องที่ว่าง



Linear Probing Hash Table

Linear Probing

หลังจากเกิดการชนครั้งที่ j ให้ใช้ h_j เพื่อคำนวณตำแหน่ง

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

ครั้งที่ 1: เลื่อนไปจากตำแหน่งจริง 1 ช่อง

ครั้งที่ 2: เลื่อนไปจากตำแหน่งจริง 2 ช่อง

ครั้งที่ 3: เลื่อนไปจากตำแหน่งจริง 3 ช่อง

ครั้งที่ 4: เลื่อนไปจากตำแหน่งจริง 4 ช่อง

$h(x) = 1$??
$h(x) = 2$??
$h(x) = 4$??
$h(x) = 6$??

0	xxx
1	null
2	xxx
3	xxx
4	xxx
5	null
6	xxx

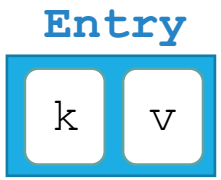
$m = 7$

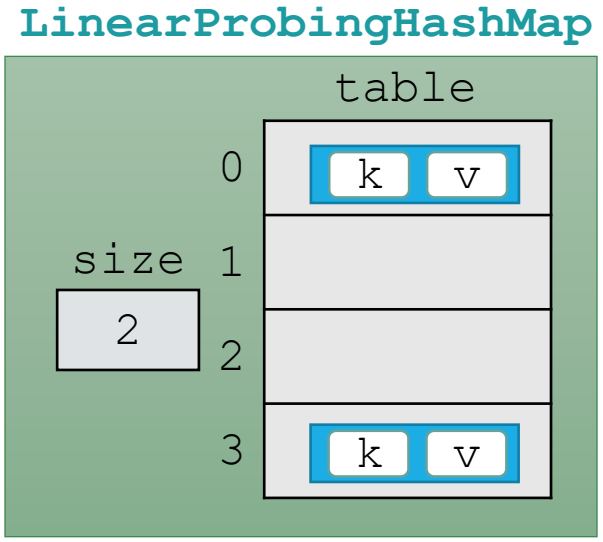
LinearProbingHashMap

```
public class LinearProbingHashMap implements Map {
   private static class Entry { ...}
   private Entry[] table;
   private int size;
   public LinearProbingHashMap(int m) {...}
   public int size() {...}
   public boolean isEmpty() {...}
   public boolean containsKey(Object key) {...}
   private int indexOf(Object key) {...}
   private int h(Object key) {...}
   public Object get(Object key) {...}
   public Object put(Object key, Object value) {...}
   public void remove(Object key) {...}
   private void rehash() {...}
}
```

LinearProbingHashMap

```
public class LinearProbingHashMap implements Map {
  private static class Entry {
    Object key, value;
    Entry(Object k, Object v) {
      key = k; value = v;
    }
  }
  private Entry[] table;
  private int size;
  public LinearProbingHashMap(int m) { table = new Entry[m]; }
  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }
  public boolean containsKey(Object key) {
    return table[indexOf(key)] != null;
  }
```



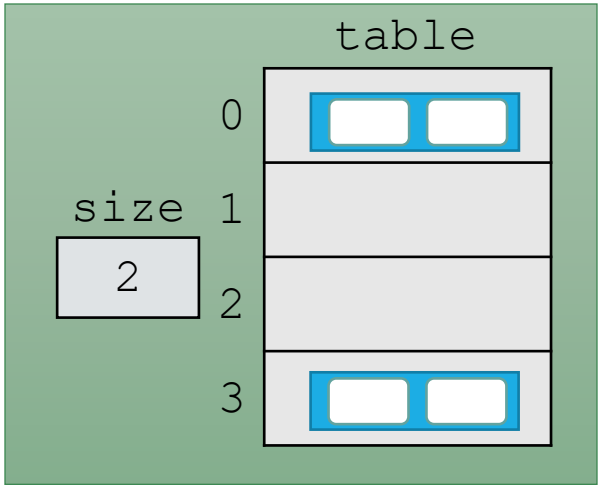


indexOf and h in LinearProbingHashMap

```
private int indexOf(Object key) {
  int h = h(key);
  for(int j=0; j<table.length; j++){ # เช็คค่าไม่เกินจำนวนช่องในตาราง
    if (table[h] == null) return h;  # เจอช่องว่าง
    if (table[h].key.equals(key))    # เจอ key นั้น
      return h;
    h = (h + 1) % table.length;      # เกิดการชน ไปดูช่องถัดไป
  }
  throw new AssertionError("No space!"); # หลุดออกมาแปลว่าไม่มีที่ว่าง
}

private int h(Object key) {
  return (key.hashCode() & 0x7FFFFFFF) % table.length;
}
```

get and put in LinearProbingHashMap

```
public Object get(Object key) {
  Entry e = table[indexOf(key)];
  return e == null ? null : e.value;
}
public Object put(Object key, Object value) {
  Object oldValue = null;
  int i = indexOf(key);
  if (table[i] == null) {
    table[i] = new Entry(key, value);
    ++size;
  } else {
    oldValue = table[i].value;
    table[i].value = value;
  }
  return oldValue;
}
```



remove in LinearProbingHashMap

```
public void remove(Object key) {
  int i = indexOf(key);
  if (table[i] != null) {
    table[i] = null;
    --size;
    for(++i; table[i]!=null; i=(i+1)%table.length) {
      Entry e = table[i];
      table[i] = null;
      table[indexOf(e.key)] = e;
    }
  }
}
```

ไล่ทุกช่องต่อจากนั้นที่มีค่าเก็บอยู่

ย้ายช่อง

Rehashing in LinearProbingHashMap

```
public Object put(Object key, Object value) {
  Object oldValue = null;
  int i = indexOf(key);
  if (table[i] == null) {                    # เจอข้อมูลที่มี key ที่ระบุ
    table[i] = new Entry(key, value);
    ++size;
    if (size > table.length/2) rehash();  # rehash เพื่อขยายตาราง
  } else {                                   # ไม่เจอข้อมูลที่มี key ที่ระบุ
    oldValue = table[i].value;
    table[i].value = value;
  }
  return oldValue;
}
```

Rehashing in LinearProbingHashMap

```
private void rehash() {
  Entry[] oldT = table;
  table = new Entry[2 * table.length];     # สร้างตารางใหม่
  for (int i = 0; i < oldT.length; i++) { # สร้างตารางใหม่
    if (oldT[i] != null)
      table[indexOf(oldT[i].key)] = oldT[i];
  }
}
```

Quadratic Probing Hash Table

Quadratic Probing

หลังจากเกิดการชนครั้งที่ j ให้ใช้ h_j เพื่อคำนวณตำแหน่ง

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

ครั้งที่ 1: เลื่อนไปจากตำแหน่งจริง 1 ช่อง

ครั้งที่ 2: เลื่อนไปจากตำแหน่งจริง 4 ช่อง

ครั้งที่ 3: เลื่อนไปจากตำแหน่งจริง 9 ช่อง

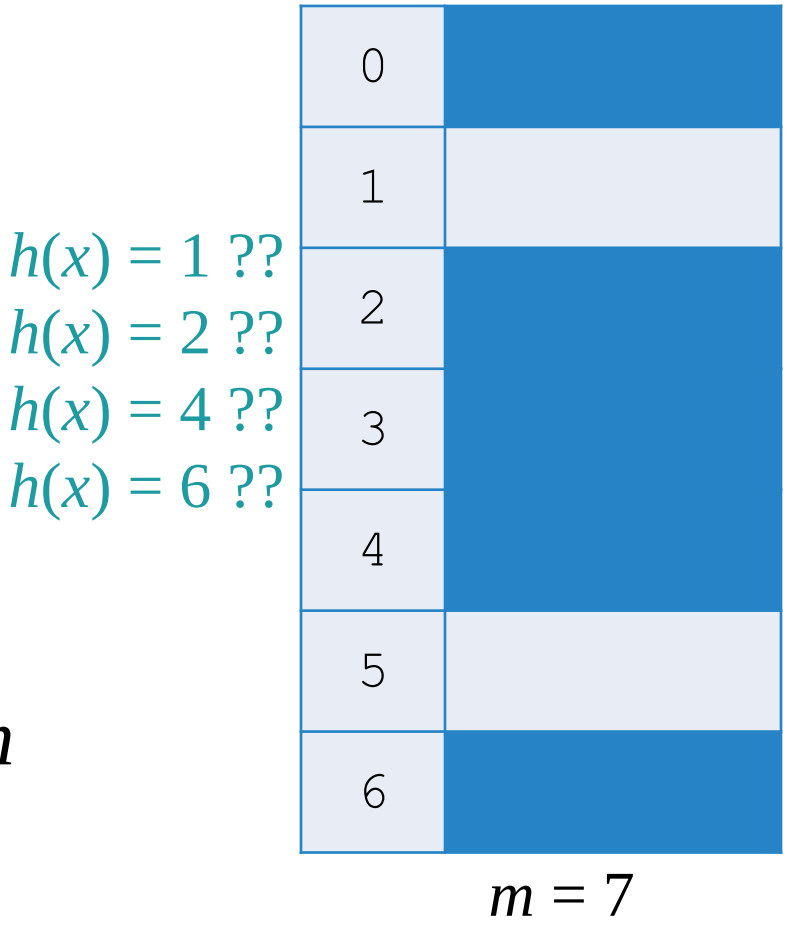
ครั้งที่ 4: เลื่อนไปจากตำแหน่งจริง 16 ช่อง

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

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

$$= (h(x) + j^2 - 2j + 1) \% m$$

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



Quadratic Probing Hash Table

```
public class QuadraticProbingHashMap implements Map {
  private static class Entry { ... }
  private static final Entry DELETED=new Entry(new Object(),null);
  private Entry[] table;
  private int size;
  private int numNonNulls;
  public QuadraticProbingHashMap(int m) {...}
  private int nextPrime(int m) {...}
  public int size() { ... }
  public boolean isEmpty() { ... }
  public boolean containsKey(Object key) { ... }
  public Object get(Object key) { ... }
  private int h(Object key) { ... }
  private int indexOf(Object key) {...}
  public Object put(Object key, Object value) {...}
  public void remove(Object key) {...}
  private void rehash() {...}
}
```

QuadraticProbingHashMap

```
public class QuadraticProbingHashMap implements Map {
  private static class Entry { ... }
  private static final Entry DELETED =
                                    new Entry(new Object(),null);
  private Entry[] table;
  private int size;
  private int numNonNulls;
  public QuadraticProbingHashMap(int m) {
    table = new Entry[nextPrime(m)];
  }
  private int nextPrime(int m) {  # return a prime after m
    BigInteger b = new BigInteger(Integer.toString(m));
    return b.nextProbablePrime().intValue();
  }
```

QuadraticProbingHashMap (cont.)

```
public int size() { ... }
public boolean isEmpty() { ... }
public boolean containsKey(Object key) { ... }
public Object get(Object key) { ... }
private int h(Object key) { ... }
private int indexOf(Object key) {
  int h = h(key);
  for (int j = 1; j < table.length; j++) {
    if (table[h] == null) break;
    if (table[h].key.equals(key)) break;
    h = (h + 2 * j - 1) % table.length;
  }
  return h;
}
```

put in QuadraticProbingHashMap

```
public Object put(Object key, Object value) {
  Object oldValue = null;
  int i = indexOf(key);
  if (table[i] == null) {
    table[i] = new Entry(key, value);
    ++size; ++numNonNulls;
    if (numNonNulls > table.length/2) rehash();
  } else {
    oldValue = table[i].value;
    table[i].value = value;
  }
  return oldValue;
}
```

remove and rehash in QuadraticProbingHashMap

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

 private void rehash() {
   Entry[] oldT = table;
   table = new Entry[nextPrime(4 * size)];
   for (int i= 0; i < oldT.length; i++) {
     if (oldT[i] != null && oldT[i] != DELETED) {
       int j = indexOf(oldT[i].key);
       table[j] = oldT[i];
     }
   }
   numNonNulls = size;
 }
```

Double Hashing

Double Hashing

```
private int indexOf(Object key) {
  int h = h(key);
  int g = g(key);
  for (int j = 1; j < table.length; j++) {
    if (table[h] == null) break;
    if (table[h].key.equals(key)) break;
    h = (h + g) % table.length;
  }
  return h;
}

private int g(Object key) {
  return 1+(key.hashCode() & 0x7FFFFFFF) % (table.length/2);
}
```

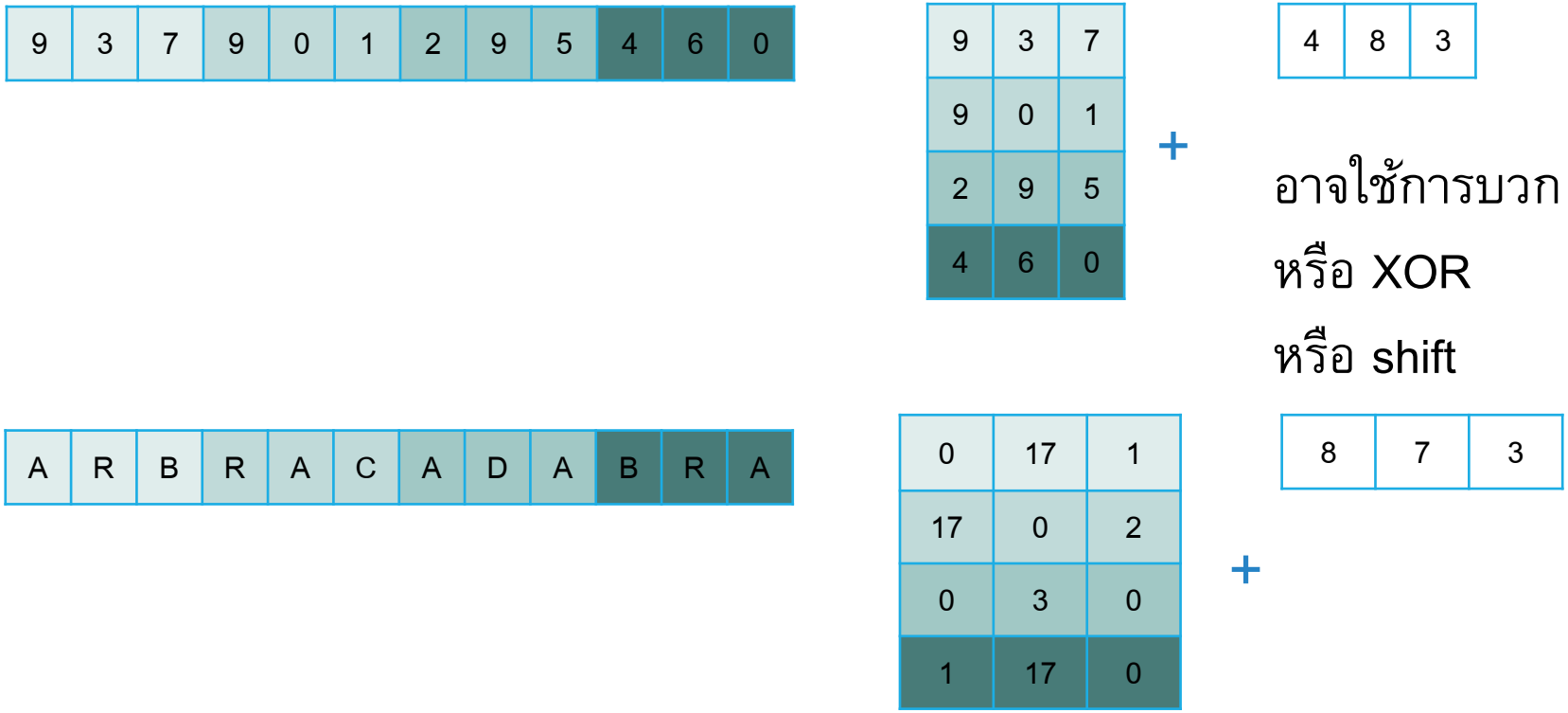
$$h_j(x) = (h(x) + j\,g(x)) \,\%\, m$$

Hash Functions

Hash function ที่ต้องการ

- กระจายข้อมูลในตารางได้ดี
- ทำให้เกิดการชนไม่บ่อย
- ตัวอย่างที่ไม่ดี ถ้าเลือกรหัสนิสิตเป็น key แล้ว hash function ตัดเอาเลข 2 หลักหลัง หรือ 3 หลักแรกของรหัสนิสิตมาใช้ จะเกิดการชนบ่อยมาก

ตัวอย่าง hash function



จาวามี method hashCode() ใน class Object

Universal Hashing

$$h(x) = \big((ax + b)\ \%\ p\big)\ \%\ m$$

x เป็นคีย์ที่เป็นจำนวนเต็มที่ไม่เป็นลบและไม่เกิน U

p เป็นจำนวนเฉพาะที่อยู่ในช่วง U ถึง $2U\text{-}1$

a, b เป็นจำนวนเต็มที่สุ่มมาและ $0 < a < p,\ 0 \leq b < p$

m เป็นขนาดของตาราง