บทที่ 9

ตารางแฮช

บทนี้เราจะมาเรียนโครงสร้างข้อมูลที่ใช้เพื่อให้หาข้อมูลได้ในเวลารวดเร็ว ซึ่งโครงสร้างข้อมูลนี้ก็คือ ตารางแฮช (hash table) ตารางนี้ใช้เก็บของเพื่อการค้นหาข้อมูลที่ต้องการอย่างรวดเร็ว ดังนั้นฟังก์ชั่นของมันจะมีแค่การเติมของ ลบของออก และหาของที่ต้องการว่าอยู่ในตารางหรือเปล่า จะไม่มีการจัดเรียงข้อมูลและการหาค่ามากหรือน้อยสุด นั่นก็คือ สรุปแล้วการเก็บข้อมูลในโครงสร้างข้อมูลนี้ ทำอะไรกับข้อมูลได้ไม่เท่าการใช้โครงสร้างต้นไม้ แต่ข้อเด่นของตารางแฮชคือ ทำให้เราสามารถหาของที่ต้องการจากตารางได้ในเวลาคงที่

รูปแบบทั่วไปของตารางแฮชที่เราจะใช้กันจะเป็นอาร์เรย์ สิ่งของที่เราจะเอาใส่ตารางแฮชจะประกอบด้วย key และ value เช่นเดียวกับ Map เราเอา key มาเข้าแฮชฟังก์ชั่น(hash function) เพื่อให้ได้ค่าดัชนีที่จะเก็บ value

ดังนั้นจุดสำคัญที่สุดของการสร้างตารางแฮช ก็คือแฮชฟังก์ชั่นนั่นเอง ซึ่งแฮชฟังก์ชั่นที่ดีควรจะมีคุณสมบัติดังนี้

- คำนวณง่าย ไม่เสียเวลา
- คีย์สองตัวต้องเข้าฟังก์ชั่นแล้วได้ค่าดัชนี(index) ของอาร์เรย์คนละตัว (อันนี้ทำได้ยาก แต่เดี๋ยวดูไปเรื่อยๆก่อนนะ)

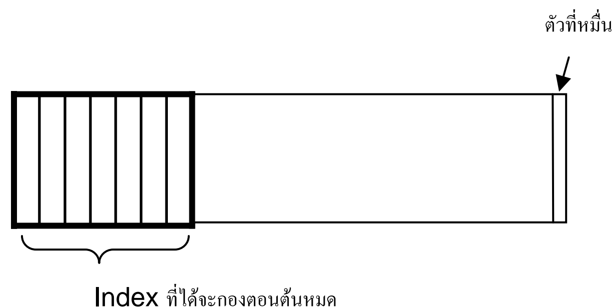
ดังนั้นผู้ออกแบบแฮชฟังก์ชั่นจะต้องพยายามให้แฮชฟังก์ชั่นกระจายคีย์ไปได้ทั่วตาราง ซึ่งเราอาจใช้ จำนวนคีย์ mod ขนาดตาราง แต่ถ้าคีย์เป็น 10, 20, 30, … ก็ใช้ฟังก์ชั่นนี้ไม่ดี ค่าคีย์จริงๆแล้วอาจจะไม่มีก็ได้ แต่คำนวณเอาจากค่า value เรามาลองดูตัวอย่างแฮชฟังก์ชั่นแบบต่างๆกัน

แบบแรกที่ผมจะแสดงให้เห็นนี้จะเป็นแฮชฟังก์ชั่นที่คิดค่าดัชนีจากการบวกค่า ASCII ของตัวอักษรที่เป็นข้อมูล(ข้อมูลแต่ละตัวให้เป็น String) โค้ดนั้นอยู่ในรูป 9.1

```
1:      public static int hash(String key, int tableSize){
2:       int hashVal = 0;
3:       for(int i =0; i<key.length(); i++)
4:          hashVal += key.charAt(i);
5:       return hashVal%tableSize;
6:      }
```

รูป 9.1 แฮชฟังก์ชั่นแบบบวกค่า ASCII

แฮชฟังก์ชั่นแบบในรูป 9.1 นี้จะมีปัญหาถ้าเรามีตารางขนาดใหญ่ เพราะถ้าคีย์ไม่ค่อยยาว เช่นมีแปดตัวอักษร เนื่องจาก ASCII มีค่าได้มากที่สุดคือ 127 ฉะนั้นผลบวกของทุกตัวอักษรจะได้ไม่เกิน 127*8 ดังนั้น ถ้าตารางใหญ่ ค่าดัชนีของอาร์เรย์ที่คิดออกมาได้จะไม่กระจาย (รูป 9.2)



รูป 9.2 จุดอ่อนของฟังก์ชั่นในรูป 9.1

เรามาลองดูแฮชฟังก์ชั่นแบบอื่นดูบ้าง สมมติว่าตารางใหญ่และตัวคีย์เป็นตัวอักษรสุ่มอย่างน้อยสามตัวอักษร เราดูเฉพาะตัวอักษรสามตัวแรก ให้ฟังก์ชั่นนั้นเป็นดังรูปที่ 9.3

```
1:      public static int hash(String key, int tableSize){
2:       return (key.charA(0)+27*key.charAt(1)+729*
3:               key.charAT(2))%tableSize;
4:      }
```

รูป 9.3 แฮชฟังก์ชั่นแบบดูตัวอักษรสามตัวแรก

ตัวเลข 27 ในรูป 9.3 นั้น คือจำนวนตัวอักษรทั้งหมด รวมถึงตัวว่างด้วย ส่วนตัวเลข 729 นั้นมาจาก 27*27 การกระจายของข้อมูลคีย์ต่างๆ นี้จะกระจายในตารางขนาดหนึ่งหมื่นได้ดี (10007 เป็นค่าแรกสำหรับหนึ่งหมื่นที่เป็น prime เดี๋ยวมาดูว่าทำไมต้องเป็น)

โค้ดในรูป 9.3 นั้นกระจายลงตารางได้ดีกว่าโค้ดในรูป 9.1 ก็จริง แต่จริงๆคีย์ที่ใช้ก็ไม่สุ่ม ดังนั้นข้อมูลที่จะถูกจัดลงตารางช่องเดียวกันต้องมีมากแน่นอน เพื่อแก้ปัญหาการลงช่องซ้ำ เราลองมาดูฟังก์ชั่นต่อไปกัน

ฟังก์ชั่นต่อไปของเราจะคำนวณฟังก์ชั่นพหุนาม(polynomial function) ของ 37 โดยใช้กฎของฮอร์เนอร์(Horner's Rule) หลายคนอาจจะไม่คุ้นกับกฎของฮอร์เนอร์ ดังนั้นเรามาดูกันก่อน เราสามารถคำนวณ k0 + 37k1+ 37*37k2 ได้โดยใช้ [(k2*37)+k1]*37 +k0 กฎของฮอร์เนอร์คือการทำแบบนี้ไป n ตัว ซึ่งจริงๆแล้วคือการคำนวณ

$$\sum_{i=0}^{KeySize-1} Key[KeySize-i-1]*37^{i}$$

นั่นเอง โค้ดของฟังก์ชั่นนี้อยู่ในรูป 9.4

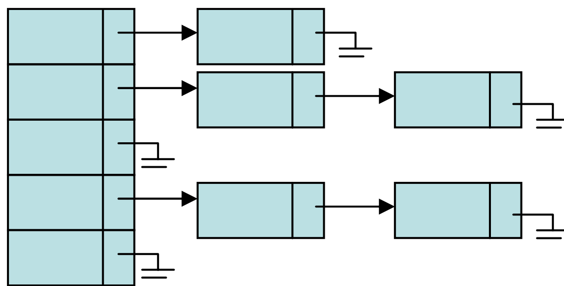
```
1:     public static int hash(String key, int tableSize){
2:      int hashVal = 0;
3:      for(int i =0; i<key.length(); i++)
4:         hashVal= 37*hashVal+key.charAt(i);
5:      hashVal %= tableSize;
6:      if(hashVal<0)
7:         hashVal += tableSize; // เผื่อ overflow
8:      return hashVal;
9:     }
```

รูป 9.4 แฮชฟังก์ชั่นจากกฎของฮอร์เนอร์

จากรูป 9.4 โค้ดอาจจะกระจายไม่ดีนัก แต่ค่อนข้างคำนวณง่าย แต่ถ้าคีย์ยาวมาก ก็จะคำนวณนาน ซึ่งเราสามารถแก้โดยไม่ใช้หมดทุกตัวอักษร อาจเลือกตัวอักษรมาบางตัวจากแต่ละส่วนที่สำคัญของคีย์นั้น แต่ยังไงแฮชฟังก์ชั่นก็แยกของลงตารางไม่ได้100% ยังไงก็จะมีของที่ลงช่องเดียวกันจนได้ เราเรียกว่าเกิดการชน (collision) ต่อไปเราจะดูวิธีแก้ปัญหาเมื่อเกิดการชน

การแก้ปัญหาการชนด้วย Separate Chaining

หลักการของวิธีนี้คือ เก็บตัวที่ซ้ำใส่ลิงค์ลิสต์ ถ้าจะหาของ ก็ใช้แฮชฟังก์ชั่นหาช่องที่ลิสต์ที่เก็บของอยู่เสียก่อนแล้วต่อด้วยการหาในลิสต์ ถ้าจะใส่ของลงไป ก็ ใช้แฮชฟังก์ชั่นเพื่อหาลิสต์ที่จะลง ต่อจากนั้นตรวจลิสต์ว่ามีของที่จะเติมลงไปหรือยัง ถ้ายังก็เติมลงไปหน้าลิสต์(ดูรูป 9.9) (ตัวที่ถูกใส่ไปใหม่ๆมักถูกเรียกใช้ในเวลาไม่นานต่อมา ดังนั้นเราจึงเอาไว้หน้าลิสต์) รูป 9.5 แสดงตารางที่ถูกสร้างขึ้นด้วยวิธีนี้



รูป 9.5 ตารางแฮชแบบใช้ Separate Chaining

โค้ดของ Separate Chaining

ตารางแฮชแบบนี้เราจะสร้างโดยใช้อินเตอร์เฟส ดังรูป 9.6

```
1:      public interface Hashable{
2:          /**
3:            * นี่คือแฮชฟังก์ชั่น
4:            * @param tableSize ขนาดของตารางแฮช
5:            * @return ตัวเลขระหว่าง 0 และ tableSize-1 กระจายกันอย่างดี
6:            */
7:          int hash( int tableSize );
8:      }
```

รูป 9.6 อินเตอร์เฟสสำหรับสร้างตารางแฮช

ตัวอย่างการใช้งานอินเตอร์เฟสนี้อยู่ในรูป 9.7 คลาสที่เราใช้งานนี้เรียกใช้งานเมธอด hash จากคลาส SeparateChainingHashTable อีกทีหนึ่ง ส่วนโค้ดของ SeparateChainingHashTable นั้นอยู่ในรูป 9.8 ถึง 9.13 โดยจะแยกเสนอเป็นส่วนๆไป จะเห็นว่าโค้ดแต่ละส่วนนั้นจะใช้

ประโยชน์จากเมธอดของลิงค์ลิสต์อยู่ไม่น้อย (ลิงค์ลิสต์ในที่นี้คือลิงค์ลิสต์ธรรมดาจากบทที่ 3 ไม่ใช่ลิงค์ลิสต์ของจาวา)

```
1:      Public class Student implements Hashable{
2:       private String name;
3:       private double number;
4:       private int year;
5:       public int hash(int tableSize){
6:          return SeparateChainingHashTable.hash(name, tableSize);
7:       }
8:
9:       public boolean equals(Object rhs){
10:         return name.equals(((Student)rhs).name);
11:      }
12:     }
```

รูป 9.7 ตัวอย่างคลาสที่ใช้งานตารางแฮช

```
1:      public class SeparateChainingHashTable{
2:         private static final int DEFAULT_TABLE_SIZE = 101;
3:         private LinkedList [ ] theLists;
4:
5:         public SeparateChainingHashTable( ){
6:            this( DEFAULT_TABLE_SIZE );
7:         }
8:
9:         /**
10:          * คอนสตรัคเตอร์
11:          * @param size ขนาดของตารางแฮชที่จะสร้าง
12:          * (อย่างคร่าวๆ เพราะจะสร้างเป็นขนาดจำนวนเฉพาะ)
13:          */
14:        public SeparateChainingHashTable(int size){
15:           theLists = new LinkedList[nextPrime(size)];
16:           for(int i = 0; i < theLists.length; i++)
17:              theLists[i] = new LinkedList();
18:        }
```

เราสร้างลิงค์ลิสต์ซึ่งแต่ละสมาชิกก็เป็นลิงค์ลิสต์ด้วย

รูป 9.8 โค้ดของ SeparateChainingHashTable ในส่วนเริ่มต้นและส่วนคอนสตรัคเตอร์

```
1:      /**
2:       * เอาของใส่ตารางแฮช ถ้าในตารางมีของแล้วก็ไม่ต้องทำอะไร
3:       * @param x ของที่จะใส่ในตาราง
4:       */
5:       public void insert(Hashable x){
6:          LinkedList whichList = theLists[x.hash(theLists.length )];
7:          LinkedListItr itr = whichList.find(x);
8:             if(itr.isPastEnd( ))
9:                 whichList.insert(x, whichList.zeroth( ));
10:      }
```

ใช้ Student ตรงนี้ไง

รูป 9.9 โค้ดของ SeparateChainingHashTable ในส่วนการ insert

```
1:      /**
2:       * การเอาของออกจากตารางแฮช
3:       * @param x ของที่จะเอาออกจากตาราง
4:       */
5:      public void remove(Hashable x){
6:         theLists[x.hash(theLists.length)].remove(x);
7:      }
```

รูป 9.10 โค้ดของ SeparateChainingHashTable ในส่วนการ remove

```
1:      /**
2:       * หาของในตาราง
3:       * @param x ของที่จะหา
4:       * @return ของที่หานั้น หรือ null ถ้าหาไม่เจอ
5:       */
6:       public Hashable find(Hashable x){
7:          return (Hashable)theLists[x.hash(theLists.length)].find(x
8:                                                   ).retrieve( );
9:       }
```

รูป 9.11 โค้ดของ SeparateChainingHashTable ในส่วนการหาว่า x อยู่ในตารางแฮชหรือไม่

```
1:      public void makeEmpty( ){
2:         for( int i = 0; i < theLists.length; i++ )
3:            theLists[ i ].makeEmpty( );
4:      }
```

รูป 9.12 โค้ดของ SeparateChainingHashTable ในส่วนการทำให้ตารางแฮชว่าง

```
1:      /**
2:       * แฮชฟังก์ชั่นสำหรับใส่สตริงเข้าตารางแฮช
3:       * @param key สตริงที่เราจะทำการใส่ตารางแฮช
4:       * @param tableSize ขนาดของตารางแฮช
5:       * @return ค่าดัชนีสำหรับเอาสตริงใส่ในตาราง
6:       */
7:       public static int hash( String key, int tableSize ){
8:          int hashVal = 0;
9:          for( int i = 0; i < key.length( ); i++ )
10:            hashVal = 37 * hashVal + key.charAt( i );
11:         hashVal %= tableSize;
12:         if( hashVal < 0 )
13:            hashVal += tableSize;
14:         return hashVal;
15:      }
```

รูป 9.13 โค้ดของ SeparateChainingHashTable ในส่วนการทำแฮช

นอกจากเมธอดต่างๆที่สำคัญที่ใช้หา เติม ลบของออกจากตารางแฮชแล้ว การใช้ตารางแฮชยังต้องคำนึงถึงขนาดตารางด้วย โดยปกติในการสร้างตารางแฮชนั้น เราจะใช้ขนาดของตารางเป็นจำนวนเฉพาะ(เหตุผลที่ใช้จำนวนเฉพาะก็เพื่อพยายามลดการชน ซึ่งจะมีตัวอย่างแสดงต่อไป) ดังนั้นจึงต้องมีเมธอดสำหรับจัดการกับจำนวนเฉพาะด้วย ซึ่งได้แสดงไว้ในรูป 9.14

```
1:      /**
2:       * หาจำนวนเฉพาะที่มากกว่าหรือเท่ากับ n
3:       * @param n ตัวเลขที่ใช้ ซึ่งต้องเป็นบวกเท่านั้น
4:       * @return จำนวนเฉพาะที่มากกว่าหรือเท่ากับ n
5:       */
6:      private static int nextPrime(int n){
7:         if(n % 2 == 0)
8:            n++;
9:         for( ; !isPrime(n); n += 2 )
10:           ;
11:        return n;
12:     }
13:
14:     /**
15:      * เมธอดใช้ทดสอบว่าตัวเลขที่ให้เป็นจำนวนเฉพาะหรือเปล่า เป็นเมธอดที่มีโค้ดไม่ค่อยมีประสิทธิภาพ
16:      * @param n ตัวเลขอินพุตที่ให้
17:      * @return ผลการทดสอบ
18:      */
19:      private static boolean isPrime(int n){
20:         if(n == 2 || n == 3)
21:            return true;
22:         if(n == 1 || n % 2 == 0)
23:            return false;
24:         for(int i = 3; i*i <= n; i+= 2)
25:            if(n%i == 0)
26:               return false;
27:         return true;
28:      }
```

รูป 9.14 เมธอดที่ใช้ในการสร้างและทดสอบตัวเลขจำนวนเฉพาะ

ส่วนตัวอย่างโปรแกรมการใช้งาน ซึ่งในที่นี้รันในเมธอดเมนนั้น อยู่ในรูป 9.15

วิเคราะห์เวลาในการทำงานของ Separate Chaining

ก่อนอื่นเราต้องเข้าใจศัพท์บางอย่างก่อน นั่นคือ Load factor (λ) ซึ่งก็คือความยาวเฉลี่ยของลิงค์ลิสต์แต่ละลิสต์ที่แบ่งใส่ของ โดยมีสูตรว่า

$$\lambda = \frac{numberOfElementsInTheTable}{tableSize}$$

การหาของในลิงค์ลิสต์นั้นกินเวลา

Search time = time to do hashing + time to search list

= constant + time to search list

ในกรณีที่หาของที่ต้องการไม่เจอ เวลาที่ใช้ก็จะเป็นเวลาที่ใช้ในการสำรวจทั้งลิสต์นั่นเอง ซึ่งก็คือสรุปได้ว่า

Unsuccessful search time = ความยาวลิสต์(เฉลี่ย) = ค่า load factor

```
1:      public static void main(String [] args){
2:         SeparateChainingHashTable H = new SeparateChainingHashTable( );
3:         final int NUMS = 4000;
4:         final int GAP  =   37;
5:
6:         System.out.println("Checking(no more output means success)" );
7:         for(int i = GAP; i != 0; i = (i + GAP)% NUMS)
8:            H.insert(new MyInteger(i)); //เอา 37 และผลคูณต่างๆของมันใส่ตารางแฮช
9:         for(int i = 1; i < NUMS; i+= 2)
10:           H.remove(new MyInteger(i)); //เอาเลขคี่ออกจากตารางให้หมด
11:        for(int i = 2; i < NUMS; i+=2)
12:           if(((MyInteger)(H.find(new MyInteger(i)))).intValue()!= i)
13:              System.out.println( "Find fails " + i ); //หาเลขคู่ i แต่เจอค่าอื่น
14:
15:        for(int i = 1; i < NUMS; i+=2){
16:           if(H.find(new MyInteger(i)) != null)
17:              System.out.println( "OOPS!!! " +  i  );
18:        } //หาเลขคี่ i แต่เจออย่างอื่น
19:     }
```

รูป 9.15 ตัวอย่างการเขียนโค้ดเพื่อใช้งานตารางแฮชในเมธอด main

ในกรณีที่หาของเจอ(Successful search) ที่ลิสต์ที่จะค้น จะมีหนึ่งโนดที่มีของและก็มีโนดอื่นๆ ซึ่งมีตั้งแต่ 0 โนดขึ้นไป ถ้าเราดูที่ตารางแฮชซึ่งมีจำนวนสมาชิก N ซึ่งถูกแบ่งไปเป็นลิสต์ M ลิสต์ จะได้ว่า

- มีโนดที่ไม่ใช่โนดที่เราต้องการหาอยู่ N-1 โนด
- ถ้าแบ่งไปให้ลิสต์ M ลิสต์ละเท่าๆกันจะได้ว่าแต่ละลิสต์มี (N-1)/M โนด
  - $=\frac{N}{M}-\frac{1}{M}=\lambda-\frac{1}{M}$
  - $=\lambda$ นั่นเอง เพราะว่า M มีค่ามาก

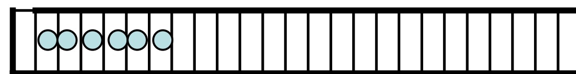
- โดยเฉลี่ยแล้ว ครึ่งหนึ่งของโนดพวกนี้จะถูกค้น นั่นคือ $\frac{\lambda}{2}$
- ฉะนั้น เวลา(จำนวนการค้น)โดยเฉลี่ยที่ใช้ในการหาของเจอคือ 1 + $\frac{\lambda}{2}$
- นี่หมายความว่า ขนาดตารางไม่สำคัญ ที่สำคัญคือ load factor หรือ λ นั่นเอง

การแก้ปัญหาการชนด้วย Open Addressing

ถ้าไม่ใช้ลิสต์ ก็ต้องใช้วิธีการนี้ หลักการคือ ถ้ามีการชน ก็หาช่องใหม่เรื่อยๆจนกว่าจะเจอช่องว่าง โดยลองช่องที่ $h_0(x), h_1(x), \ldots$ ซึ่ง $h_i(x) = hash(x) + f(i)\% tableSize$, $f(0) = 0$ ข้อมูลทุกตัวนั้นต้องลงตาราง ไม่มีลิสต์มาช่วยเก็บ ดังนั้นตารางต้องใหญ่ (Load factor <=0.5) การหาช่องใหม่นั้น มีหลายวิธีด้วยกัน

การหาช่องใหม่ด้วย Linear Probing

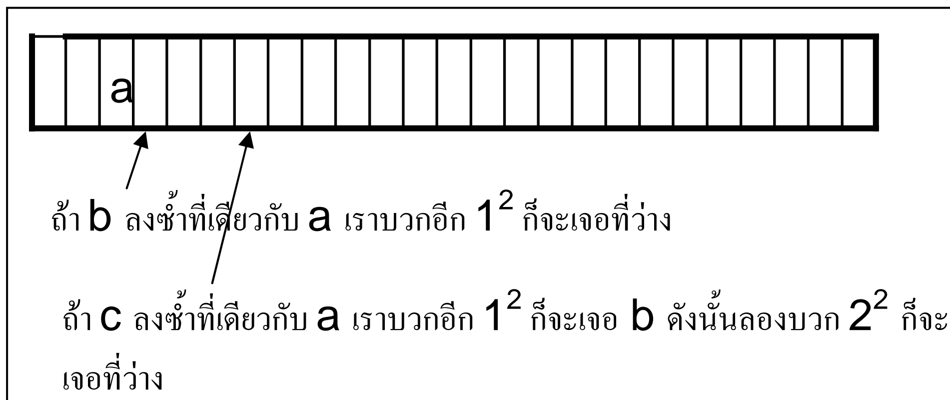
วิธีการนี้ ให้ f เป็นลิเนียร์ฟังก์ชั่นของ i นั่นคือ f(i)=i ซึ่งสรุปแล้วคือการหาช่องว่างในตารางโดยดูเรียงทีละช่องไปเรื่อยๆ วิธีนี้แม้เรียบง่าย แต่ก็ทำให้เกิดปัญหาได้คือ เวลาในการหาช่องว่างจะนานได้ เพราะในการหาแต่ละครั้ง ต้องเริ่มจากการบวกเพิ่มด้วย f(1), f(2), f(3)…นั่นคือ 1,2,3,… เรียงไปเรื่อยๆจากจุดที่บอกด้วยแฮชฟังก์ชั่นทั้งสิ้น เนื่องจากเราหาเรียงทีละช่อง ดังนั้นของที่เอาใส่ตารางแฮชจะเกิดการเกาะกลุ่มในช่องติดกันอย่างมาก(เรียกว่า primary clustering) ทำให้ถ้าเกิดการชนในช่องหนึ่ง โอกาสที่ไปช่องถัดไปแล้วไม่เจอว่ามีของอยู่แล้วจะมีน้อยทีเดียว สังเกตรูป 9.16 เป็นตัวอย่าง



รูป 9.16 Primary Clustering

การหาช่องใหม่ด้วย Quadratic Probing

วิธีนี้ช่วยกำจัด primary clustering โดยฟังก์ชั่นที่นิยมใช้กันคือ $f(i) = i^2$ โดย $h_i(x) = hash(x) + f(i)\% tableSize$ เหมือนเดิม รูป 9.17 แสดงตัวอย่างการลง a, b, c ลงในตารางตามลำดับ โดยสมมติว่าลงไปแล้วแฮชฟังก์ชั่นให้ลงช่องเดียวกับ a ทุกครั้ง



รูป 9.17 การลง a, b, c ในตารางแล้วได้ช่องซ้ำกัน จึงต้องเลื่อน

จากรูป 9.17 จะเห็นว่าการลงช่องซ้ำจะก่อให้เกิดการหาช่องใหม่ในระยะกว้างออกไปเรื่อยๆ ถ้าใช้ตารางยังไม่ถึงครึ่งหนึ่งและขนาดตารางเป็นจำนวนเฉพาะแล้วละก็ ได้มีการพิสูจน์แล้วว่าหาช่องลงได้แน่นอน แต่ถ้าตารางใส่เกินครึ่งหนึ่งแล้ว หรือถ้าขนาดตารางไม่เป็นจำนวนเฉพาะ สูตรนี้ก็ไม่แน่ว่าจะหาช่องว่างได้

การพิสูจน์ว่าหาช่องลงได้แน่นอน

อย่างที่ได้กล่าวไปแล้ว ถ้าใช้ตารางยังไม่ถึงครึ่งหนึ่งและขนาดตารางเป็นจำนวนเฉพาะแล้วละก็ เราสามารถหาช่องลงได้แน่นอน ในส่วนนี้เราจะมาพิสูจน์กัน

ให้ขนาดตารางเป็นจำนวนเฉพาะที่ใหญ่กว่าสาม เราสมมติให้มี $(hash(x) + i^2)\% tableSize$ และ $(hash(x) + j^2)\% tableSize$ เป็นสองตำแหน่งที่หาที่ลงได้ โดย $0 \le i, j \le \left\lfloor \frac{tableSize}{2} \right\rfloor$

เราพิสูจน์ด้วยความขัดแย้ง (Prove by Contradiction) โดย สมมติให้ตำแหน่งทั้งสองเป็นตำแหน่งเดียวกันและ i ไม่เท่ากับ j

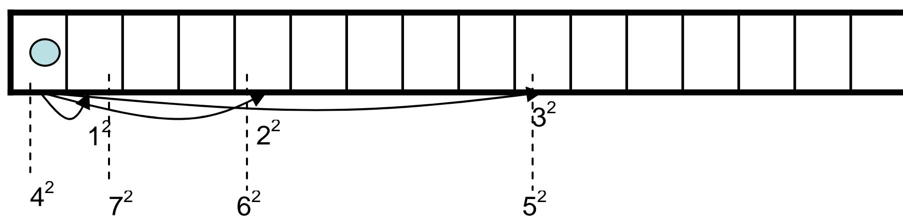
เราจะได้สมการออกมาดังนี้

$$hash(x) + i^2 = hash(x) + j^2$$
$$i^2 = j^2$$
$$i^2 - j^2 = 0$$
$$(i - j)(i + j) = 0$$

i-j =0 นั้นเป็นไปไม่ได้ เพราะเราสมมติแล้วว่าสองตัวนี้ไม่เท่ากัน ส่วน i+j=0 ก็เป็นไปไม่ได้ เพราะสมมติว่า $0 \le i, j \le \left\lfloor \frac{tableSize}{2} \right\rfloor$ ไปแล้ว ดังนั้นที่พึ่งสมมติให้ ตำแหน่งที่ลงได้ เป็นตำแหน่งเดียวกัน จึงผิด สรุปคือ ตำแหน่งที่ลงได้ เป็นคนละตำแหน่งเสมอ เราจึงพูดได้ว่าถ้าตารางยังมีการใช้งานไม่ถึงครึ่ง จะต้องได้ช่องลงแน่นอน

ขนาดตารางต้องเป็นจำนวนเฉพาะ

ถ้าขนาดตารางไม่เป็นจำนวนเฉพาะ ตำแหน่งที่ควรจะลงได้จะลดลงมากทีเดียว รูป 9.18 แสดงตารางขนาด 16 ช่อง สมมติว่าการแฮชธรรมดาให้ลงช่องแรก พอเกิดการชน quadratic probing จะทำการกระโดดหาช่องแล้วซ้ำช่องที่เคยมีของแล้วบ่อยมาก แต่ถ้าให้จำนวนช่องเป็นจำนวนเฉพาะ ก็จะทำให้การกระโดดหาช่องได้ช่องซ้ำเดิมไม่มากนัก (เลขกำกับ แสดงค่าของ i^2)



รูป 9.18 ตารางแฮชที่มีขนาดไม่เป็นจำนวนเฉพาะ

การลบข้อมูลออกจากตารางแฮชแบบ Open Addressing

เราไม่สามารถใช้การลบแบบธรรมดาได้ สำหรับเหตุผลว่าทำไมนั้น โปรดดูรูป 9.18 ในการหาของในตารางแฮช ถ้าไปยังตำแหน่งที่ควรจะมีของแล้ว ปรากฏว่า ช่องตารางช่องนั้นว่าง โดยทั่วไปเราอาจถือว่า ไม่มีของอยู่แม้จะหาแล้ว จากแนวคิดนี้ ถ้าเอาตัววงกลมในรูป 9.18 ออกจากตารางแฮช พอทีหลังหาสมาชิก(ที่ตอน insert ต้องกระโดดหลบช่องที่วงกลมอยู่) จะกลายเป็นเจอช่องว่างก่อนที่จะเจอตำแหน่งที่ถูกต้อง ซึ่งจะมีความหมายเหมือนการหาไม่เจอ

เพื่อเป็นการกันการเข้าใจผิดจากกรณีดังกล่าว แทนที่เราจะลบของโดยลบออกจากช่องในตารางแฮชเลย ใช้วิธีทำเครื่องหมายที่ช่องนั้น(แทนที่จะทำให้ช่องนั้นว่าง ก็ให้ใส่เครื่องหมายพิเศษลงไปแทน) พอตรวจช่องนั้นแล้วเจอเครื่องหมาย จะได้ถือว่ามีของอื่นอยู่ แล้วจะได้หาด้วยการกระโดดไปดูในช่องถัดไป ดังนั้นจะไม่เข้าใจผิดว่าไม่มีของแล้วทั้งที่ยังตรวจต่อได้อยู่ วิธีการลบแบบไม่ได้ลบจริงๆนี้เรียกว่า การลบแบบเกียจคร้าน(lazy deletion)

โค้ดตัวอย่างของ Open Addressing

เราใช้สิ่งที่เหมือนกับ Hashable ทุกอย่าง แต่ว่ามีการทำเครื่องหมายว่าสมาชิกตัวนี้ถูกลบออกไปจากตารางแฮชหรือยังด้วย เราเรียกคลาสใหม่ที่เพิ่มส่วนนี้เข้ามาว่า HashEntry ซึ่งดูได้ตามรูป 9.19

```
1:     class HashEntry {
2:             Hashable element;    // ตัว Hashable ที่ใช้ตามปกติ
3:             boolean  isActive;   // เป็นค่า false ถ้าถูกลบไปแล้ว
4:
5:             public HashEntry(Hashable e){
6:                 this(e, true);
7:             }
8:
9:             public HashEntry(Hashable e, boolean i){
10:                element    = e;
11:                isActive   = i;
12:            }
13:     }
```

รูป 9.19 คลาส HashEntry

โค้ดตัวอย่างที่เหลือนั้นเป็นโค้ดของตารางแฮชแบบ Quadratic Probing โดยส่วนของตัวแปรและคอนสตรัคเตอร์อยู่ในรูป 9.20

```
1:      public class QuadraticProbingHashTable{
2:         private static final int DEFAULT_TABLE_SIZE = 11;
3:
4:         /** อาร์เรย์ของสมาชิกที่เราแฮชได้ */
5:         private HashEntry [ ] array;    // ตัวอาร์เรย์
6:         private int currentSize;        // จำนวนช่องอาร์เรย์ที่มีของ
7:
8:         public QuadraticProbingHashTable( ){
9:            this( DEFAULT_TABLE_SIZE );
10:        }
11:
12:        /**
13:         สร้างตารางแฮช
14:         @param size ขนาดของตารางในตอนเริ่มต้น
15:        */
16:        public QuadraticProbingHashTable(int size){
17:           allocateArray(size);
18:           makeEmpty( );
19:        }
```

รูป 9.20 ตารางแฮชแบบ Quadratic Probing ส่วนนิยามตัวแปรและคอนสตรัคเตอร์

จากรูป 9.20 จะเห็นว่าต้องมีการเรียกใช้เมธอด allocateArray และ makeEmpty ดังนั้นเรามาดูโค้ดของสองเมธอดนี้กันในรูป 9.21

```
1:      /**
2:       * เมธอดสำหรับเตรียมตารางแฮช
3:       * @param arraySize ขนาดของตารางแฮช
4:       */
5:       private void allocateArray(int arraySize){
6:          array = new HashEntry[arraySize];
7:       }
8:
9:       /**
10:       * ทำให้ตารางแฮชว่าง
11:       */
12:       public void makeEmpty( ){
13:          currentSize = 0;
14:          for( int i = 0; i < array.length; i++ )
15:             array[ i ] = null;
16:       }
```

รูป 9.21 เมธอด allocateArray และ makeEmpty ของตารางแฮชแบบ Quadratic Probing

รูป 9.22 แสดงเมธอด isActive ซึ่งใช้ตรวจสอบว่า ในตำแหน่งที่เราสนใจนั้น มีของอยู่จริงๆ หรือไม่ พูดง่ายๆคือ ที่ตำแหน่งนั้นของตารางแฮชต้องไม่เป็น null และของข้างในต้องไม่ได้ถูกทำเครื่องหมายว่าถูกลบไปแล้ว

```
1:      /**
2:       * รีเทิร์น true ถ้าcurrentPos มีอยู่จริง และต้องมีค่าisActive เป็นจริงด้วย
3:       * @param currentPos ผลจากการเรียกเมธอด findPos
4:       * @return true ถ้าcurrentPos มีของอยู่จริง
5:       */
6:       private boolean isActive(int currentPos){
7:          return array[currentPos] != null &&
8:                                  array[currentPos].isActive;
9:       }
```

รูป 9.22 เมธอด isActiveของตารางแฮชแบบ Quadratic Probing

การหา x ในตารางแฮช ใช้เมธอด findPos ซึ่งจะหา x จากตารางแฮชจนกว่าจะเจอ null หรือเจอ x แต่ว่า x ที่เจอนั้นอาจจะถูกลบออกไปด้วยการลบแบบ lazy deletion แล้วก็ได้ ดังนั้นจึงต้องมีเมธอด find เพื่อหาว่าของในตำแหน่งที่หาเจอด้วย findPos นั้นถูกทำเครื่องหมายว่าลบออกไปจริงหรือไม่ หรือเป็น null หรือไม่ โดยใช้เมธอด isActive ทดสอบ ถ้าไม่เป็น null และไม่ถูกทำเครื่องหมายว่าลบออกไป ให้รีเทิร์นของนั้น แต่ถ้ามิเช่นนั้นให้รีเทิร์น null โค้ดของเมธอด findPos และ find นั้นอยู่ในรูป 9.23

```
1:      /**
2:       * กระโดดหาของโดยใช้quadratic probing
3:       * @param x ของที่เราต้องการหา
4:       * @return ตำแหน่งที่หาแล้วเจอnull หรือ เจอx
5:       */
6:       private int findPos(Hashable x){
7:          int collisionNum = 0;
8:          int currentPos = x.hash(array.length);
9:          while(array[currentPos] != null &&
10:                !array[currentPos].element.equals(x)){
11:             currentPos += 2 * ++collisionNum - 1;
12:             if(currentPos >= array.length) // Implement the mod
13:                currentPos -= array.length;
14:          }
15:          return currentPos;
16:       }
17:
18:     /**
19:       * หาของในตารางแฮช แล้วตรวจด้วย ว่ามีจริงหรือไม่
20:       * @param x ของที่เราต้องการหา
21:       * @return สิ่งที่หาเจอ
22:       */
23:       public Hashable find(Hashable x){
24:          int currentPos = findPos(x);
25:          return isActive(currentPos)?array[currentPos].element: null;
26:       }
```

$f(i)=i^2= (i-1)^2 +2i-1$
$= f(i-1)+2i-1$

รูป 9.23 เมธอด findPos และ find ของตารางแฮชแบบ Quadratic Probing

เมธอดต่อมาคือ เมธอด insert ซึ่งใช้ใส่ของเข้าไปในตารางแฮช โดยตอนแรกจะพยายามหาที่ให้ของนั้นลงด้วยเมธอด findPos ซึ่งถ้าเจอของอยู่แล้ว (ต้อง active ด้วย) ก็ไม่ต้องทำอะไร แต่ถ้าไม่เจอของ (หรือเจอแต่ว่าถูกตั้งค่าเป็น inactive) ก็ต้องใส่ของเข้าไปใหม่โดยคราวนี้ตั้งค่าให้ active ด้วย นอกจากนี้ถ้าของที่ใส่เข้าไปทำให้ตารางแฮชเต็มเกินครึ่งหนึ่ง ก็ต้องทำการขยายตารางแฮชด้วยเมธอด rehash (ที่ต้องขยายก็เพื่อให้แน่ใจว่า จะมีที่ให้ของลงเสมอ ตามที่ได้พิสูจน์ไว้เรื่องการหาช่องลงที่แน่นอน) เมธอด insert นั้นอยู่ในรูป 9.24 ส่วนเมธอด rehash นั้นอยู่ในรูป 9.25

```
1:      /**
2:       * เอา x  ใส่ตารางแฮช ถ้าในตารางมี x  ที่ active  อยู่แล้วก็ไม่ต้องทำอะไร
3:       * @param x  ของที่จะใส่ลงในตาราง
4:       */
5:       public void insert(Hashable x){
6:           int currentPos = findPos(x);
7:           if(isActive(currentPos))
8:               return; //x  อยู่ในตารางและ active  ด้วย กรณีนี้ไม่ต้องทำอะไร
9:           array[currentPos] = new HashEntry(x, true); //กรณีอื่น ใส่ x  ลงในตาราง
10:
11:          // Rehash  เมื่อตารางขยายเกินขนาดที่เราได้พิสูจน์ว่าหาช่องให้ของลงได้แน่
12:          if( ++currentSize > array.length / 2)
13:              rehash( );
14:      }
```

รูป 9.24 เมธอด insert ของตารางแฮชแบบ Quadratic Probing

```
1:      private void rehash( ){
2:          HashEntry [ ] oldArray = array;
3:
4:          //สร้างตารางแฮชใหม่ที่มีขนาดมากกว่าเดิมสองเท่า
5:          allocateArray(nextPrime(2 * oldArray.length));
6:          currentSize = 0;
7:
8:          //ก๊อปปี้ตาราง
9:          for(int i = 0; i < oldArray.length; i++)
10:             if(oldArray[i] != null && oldArray[i].isActive)
11:                 insert(oldArray[i].element);
12:         return;
13:     }
```

ต้องคำนวณตำแหน่งการใส่ใหม่เพราะนี่มันคนละตาราง

รูป 9.25 เมธอด rehash ของตารางแฮชแบบ Quadratic Probing

การขยายตารางโดยใช้การรีแฮชนั้น ทำได้สามลักษณะคือ

- ทำทันทีที่ตารางเต็มครึ่งหนึ่ง
- ทำเมื่อ insert ไม่ได้แล้วเท่านั้น หรือ

- ทำเมื่อถึงโหลดแฟกเตอร์ค่าหนึ่ง

สิ่งที่เราควรคำนึงถึงก็คือ อย่าลืมว่ายิ่งโหลดแฟกเตอร์มาก ก็จะ insert ได้ช้าลง ถ้าเลือกโหลดแฟกเตอร์ให้ดีๆเราก็สามารถทำการรีแฮชได้อย่างมีประสิทธิภาพ

ต่อไปเป็นเมธอดสุดท้ายของตารางแฮชแบบ Quadratic Probing นั่นก็คือเมธอดที่ใช้เอาของออกจากตาราง หรือ remove นั่นเอง ซึ่งเมธอดนี้อยู่ในรูป 9.26

```
1:      /**
2:       * เอาของออก โดยใช้การทำเครื่องหมายว่า "เอาออกแล้ว"
3:       * @param x ของที่เราจะเอาออกจากตารางแฮช
4:       */
5:       public void remove(Hashable x){
6:          int currentPos = findPos(x);
7:          if(isActive(currentPos))
8:             array[currentPos].isActive = false;
9:       }
```

รูป 9.26 เมธอด remove ของตารางแฮชแบบ Quadratic Probing

เมธอด hash, nextPrime, isPrime นั้นเหมือนเดิม ดังนั้นเราจะไม่พูดถึงอีก

การแฮชสองชั้น(Double Hashing)

ในการทำ Quadratic Probing นั้น แม้ว่าจะไม่มีปัญหาข้อมูลไปกระจุกที่เดียวในตาราง (Primary Clustering) แต่ก็ทำให้ข้อมูลไปกระจุกอยู่เป็นระยะๆในตาราง (ตามการกระโดดหาช่องว่าง) ปัญหาการกระจุกตัวแบบนี้เรียกว่า Secondary Clustering ซึ่งสามารถแก้ได้โดยการทำการแฮชสองชั้น (Double Hashing) ซึ่งในการทำนั้นจะใช้ฟังก์ชั่นดังนี้

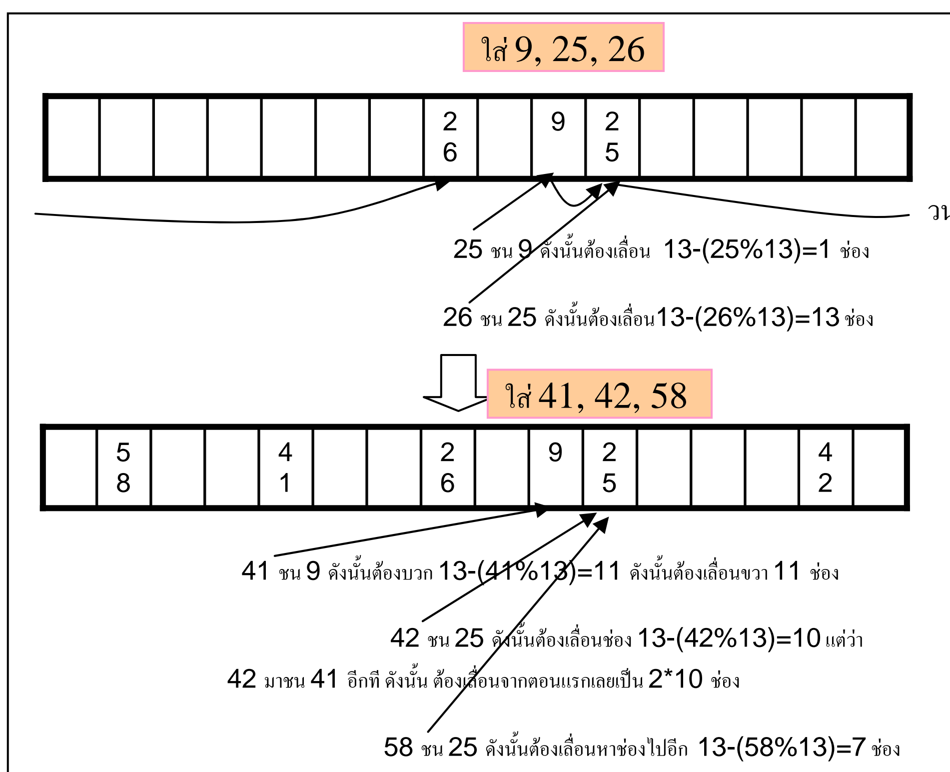
$h_i(x) = hash(x) + f(i) \% tableSize$ โดย $f(i) = i * hash_2(x)$

จะเห็นว่า ระยะที่กระโดดไปจากการหาแต่ละครั้งนั้นเท่ากับ $hash_2(x)$ ดังนั้น จึงต้องเลือกฟังก์ชั่นการแฮชฟังก์ชั่นที่สองนี้ให้ดี ตัวอย่างที่ไม่ดีเช่น

- ถ้ามีเก้าช่องและเอา $hash_2(x)$ เป็น $x\%9$ ก็จะไม่ได้อะไรเลยตอนเราใส่ 99 เพราะเป็น 0 ตลอด

นอกจากจะต้องระวังมิให้ฟังก์ชั่น $hash_2(x)$ รีเทิร์นค่าเป็น 0 แล้ว เรายังต้องให้แน่ใจว่าสามารถเข้าถึงทุกช่องในตารางได้

รูป 9.27 แสดงตัวอย่างของ $hash_2(x)$ โดยให้ $hash_2(x) = R-(x\%R)$ เมื่อ R เป็นจำนวนเฉพาะที่น้อยกว่า tableSize สมมติว่า hash อันแรกเป็น $x\%tableSize$ เราลองกับตารางขนาด 16 ทำการใส่ 9, 25, 26, 41, 42, 58 ลงไปตามลำดับ



รูป 9.27 ตัวอย่างการใช้แฮชสองชั้น

แบบฝึกหัด

1. ในการแฮชเอาจำนวนเต็มใส่ตาราง เราให้แฮชฟังก์ชั่นเป็น hash(x) = x mod 17 และใช้วิธีการเลี่ยงการชนโดยเอา hash(x) + (i*$hash_2$(x)), i เป็นจำนวนเต็มตั้งแต่ 1 ขึ้นไป ถ้าเราให้ $hash_2$(x) = 13 – (x mod 13) และขนาดของตารางเป็น 17 (มีค่าดัชนี ตั้งแต่ 0 ถึง 16) ถามว่า เมื่อเราใส่ 20, 37, 54, 19, 36, 53 ตามลำดับ ลงในตารางที่ว่าง ค่าดัชนีในตารางของจำนวนเหล่านี้จะเป็นเท่าใด

2. กำหนดให้บริษัทขายรถมือสองแห่งหนึ่งต้องการจัดเก็บข้อมูลรถยนต์ที่มีอยู่เพื่อให้สามารถสืบค้นได้ง่ายและรวดเร็ว โดยข้อมูลรถยนต์จะประกอบด้วยรายการต่อไปนี้
   - หมายเลขทะเบียนรถยนต์ – ใช้รูปแบบ xx yyyy z โดยที่ xx คือตัวอักษรไทย2 หลัก yyyy คือตัวเลข 4 หลัก และ z คือชื่อจังหวัดในประเทศไทย
   - ปีที่ผลิต – ใช้รูปแบบปี พ.ศ.
   - ประเภทรถยนต์ – มี 4 ประเภทคือ รถยนต์ส่วนตัว รถกระบะ รถตู้
   - สีรถ – มี 5 สีคือ สีขาว สีดำ สีบรอนซ์ สีน้ำเงิน
   - ราคา – มีหน่วยเป็น บาท

จงเลือกกำหนดขนาดตารางแฮชที่ใช้ในการจัดเก็บข้อมูลรถยนต์ของบริษัทที่มีอยู่ไม่เกิน 100,000 คัน และออกแบบแฮชฟังก์ชั่น H(key) โดยที่พิจารณาทั้งหมายเลขทะเบียนรถยนต์และปีที่ผลิต เป็น key

a) ตารางแฮชมีขนาดเท่าไร จึงจะเหมาะสมที่สุด เพราะเหตุใด
b) แฮชฟังก์ชั่น H(key) ควรเป็นอย่างไรตามที่ได้เรียนมาโดยให้มีการชนน้อยที่สุดและไม่เกินขนาดตาราง
c) จงเขียนส่วนของโปรแกรมภาษาจาวา ของคลาส Car โดยให้มีสิ่งต่อไปนี้ให้ครบถ้วน (นิสิตจะ implement ตารางแฮชตามที่เรียนมา)
   - Attribute ของคลาส Car ครบตามที่กำหนดไว้ในโจทย์
   - Constructor ของคลาส Car
   - เมธอด int hash() ที่ใช้เป็นแฮชฟังก์ชั่นที่กำหนดไว้ในข้อข้างบน

3. ในการจัดเก็บข้อมูลแบบตารางแฮชตามที่ได้เรียนมา เราสามารถกำหนดขนาดของตารางแฮชได้ตามต้องการ สมมติว่าระบบที่เราใช้มีข้อกำหนดว่า การจองอาร์เรย์แต่ละครั้ง จะได้อาร์เรย์ที่มีขนาด 1024 ช่องเสมอ จงเสนอโครงสร้างข้อมูลแบบตารางแฮชสำหรับเก็บข้อมูลในระบบดังกล่าว โดยให้ใช้เนื้อที่น้อยและทำงานได้เร็ว และอธิบายการทำงานของการ insert, find, remove
4. ถ้าเรามีแฮชฟังก์ชั่น hash(x) = x mod tableSize และ hash2(x) = R – x mod R เมื่อ R เป็นจำนวนเฉพาะที่น้อยกว่า tableSize จงวาดรูปผลการใส่ 7,20,21,33,34,47 เข้าไปในตารางแฮชแบบ quadratic probing และ double hashing ทั้งสองวิธีมีจำนวนครั้งการรันฟังก์ชั่นการแก้การชนต่างกันกี่ครั้ง