

ตารางแฮช

(Hash Tables)

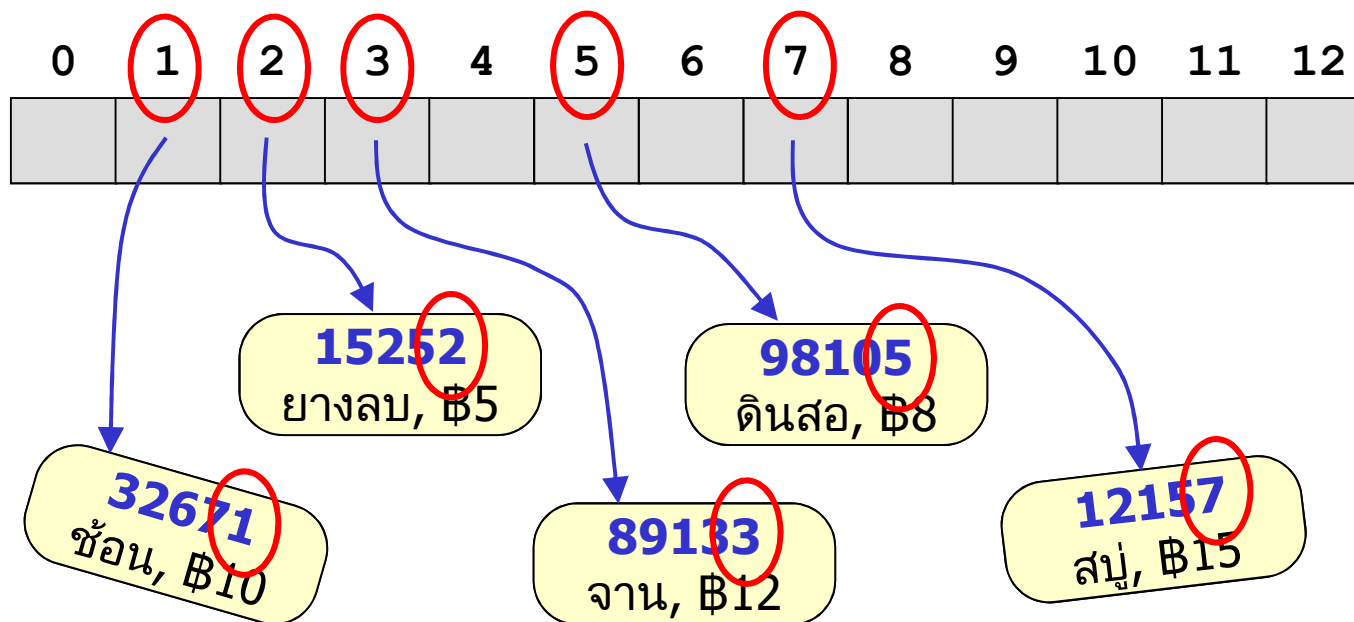
สมชาย ประสิทธิ์จตุระกุล

หัวข้อ

- การใช้ตารางเก็บข้อมูลด้วยฟังก์ชันดัชนี
- การเก็บข้อมูลแบบแยกกันโยง
- ฟังก์ชันแฮช
- กลวิธีการเขียนฟังก์ชันแฮช
- การแฮชใน C++
- การกำหนดเลขที่อยู่เปิด
- การเกาะกลุ่มของข้อมูล

ใช้ฟังก์ชันดัชนีคำนวณตำแหน่ง

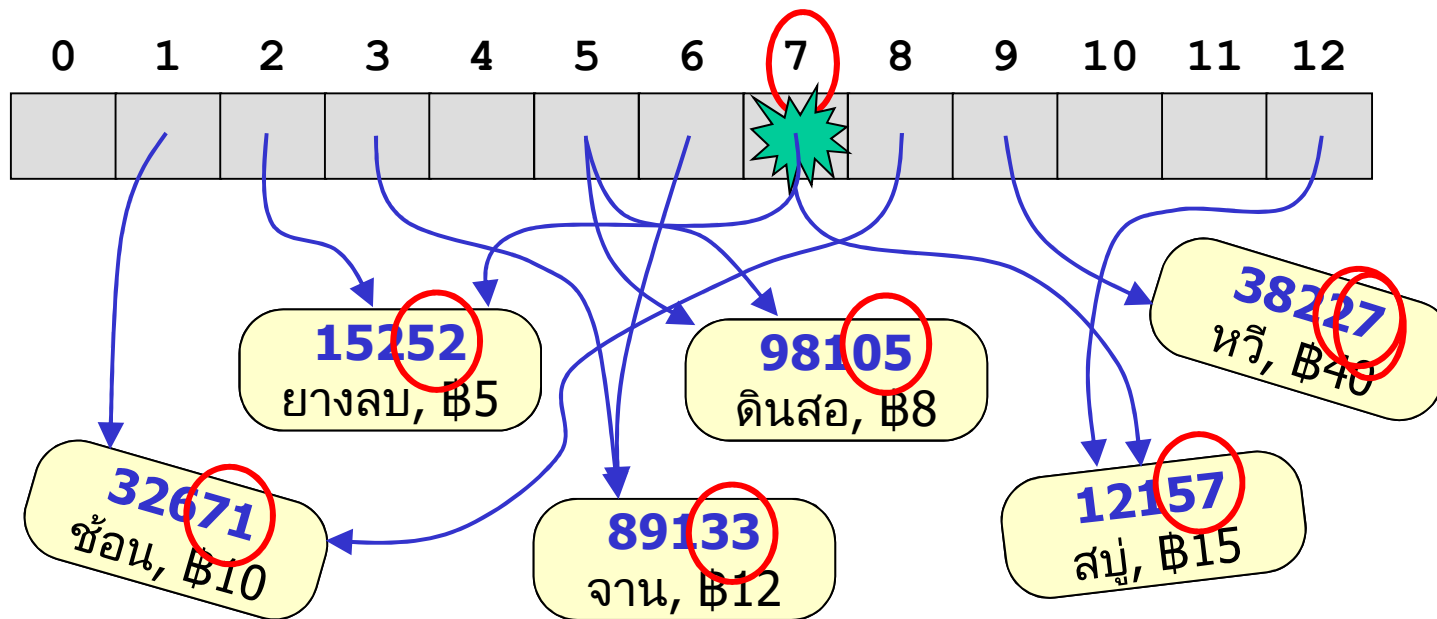
- key ของข้อมูลคือส่วนของข้อมูลที่ใช้ในการค้น
- มีตารางซึ่งแต่ละช่องเป็นที่เก็บข้อมูล
- หา $f(\text{key})$ เพื่อแปลง key ไปเป็น index ของตาราง
- ฟังก์ชันดัชนีหาไม่ยาก ถ้าองตารางขนาดใหญ่ ๆ



$$f(\text{key}) = \text{key} \% 10$$

ฟังก์ชันดัชนีนั้นหายาก

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



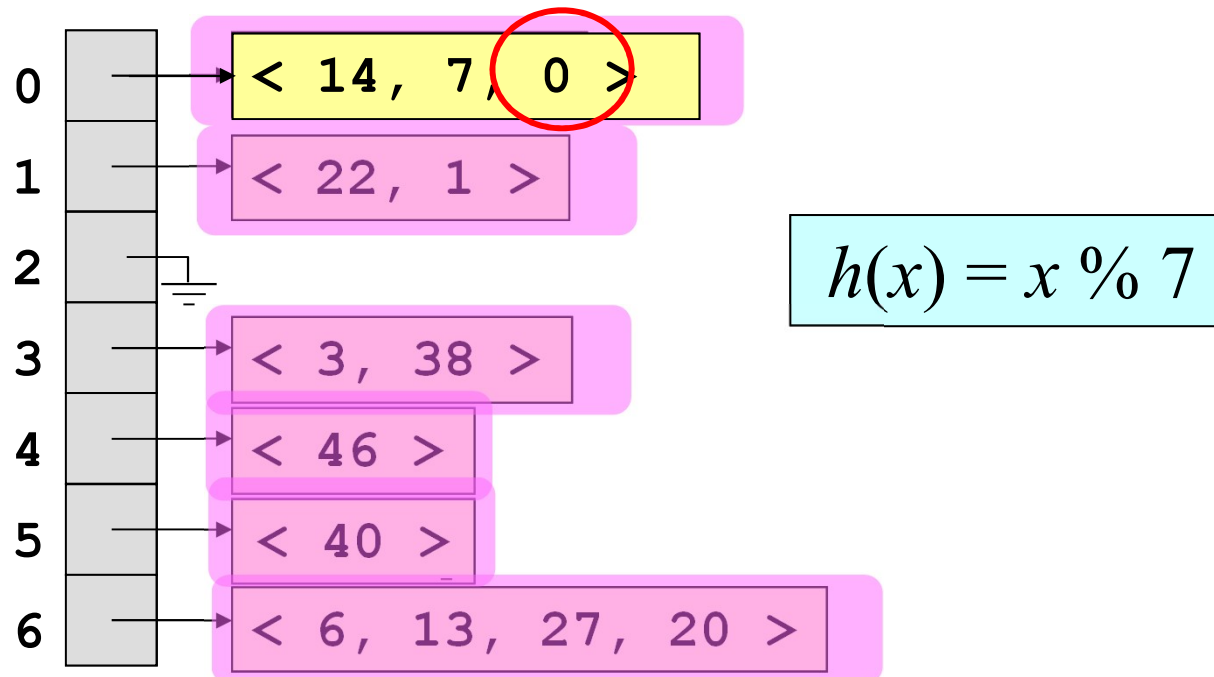
$$f(\text{key}) = f(\text{key} / 10) \oplus \text{key} \% 10$$

เปลี่ยนกลยุทธ์ : อนุญาตให้ชนได้

- จะได้เก็บข้อมูลในตารางที่ไม่ใหญ่มาก
- แต่ต้องหาวิธีแก้ไขปัญหาคารชน ที่ทำงานได้เร็ว ๆ

Separate Chaining

- จัดเก็บกลุ่มข้อมูลที่ชนกันไว้ในรายการเดียวกัน



การกระจายของข้อมูล

- ถ้าข้อมูลกระจายทั่วตาราง
 - แต่ละช่องเก็บรายการยาว $\approx \lambda$
 - ถ้า λ น้อย ค้นหาได้เร็ว
- ถ้าไม่กระจาย
 - มีบางรายการยาวเกิน λ มาก
 - การค้นหาช้าเหมือนเก็บด้วย list

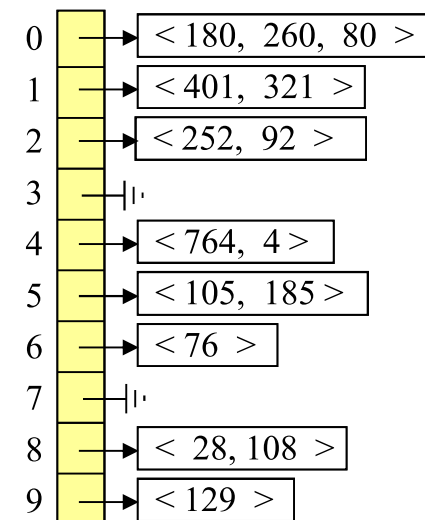
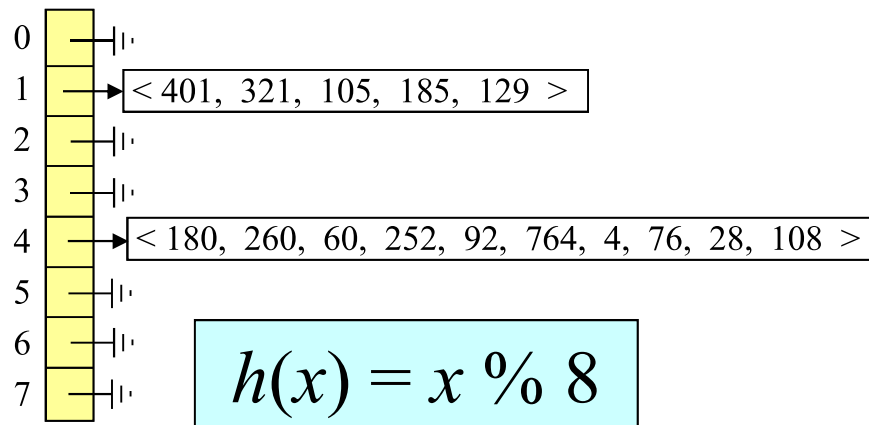
load factor

$$\lambda = n / m$$

ปริมาณ
ข้อมูล

ขนาดของ
ตาราง

$$h(x) = x \% 10$$



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
 - ลับ, เหลก, นำมาโขลกเข้าด้วยกัน



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

ตัวอย่างฟังก์ชันแฮช

```
size_t h1(size_t x) {  
    return (2654435769U * x) >> 22;  
}
```

```
size_t h2(size_t x) {  
    x = ~x + (x << 15);  
    x ^= (x >> 11);  
    x += (x << 3);  
    x ^= (x >> 5);  
    x += (x << 10);  
    x ^= (x >> 16);  
    return x & 0x3FF;  
}
```

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)

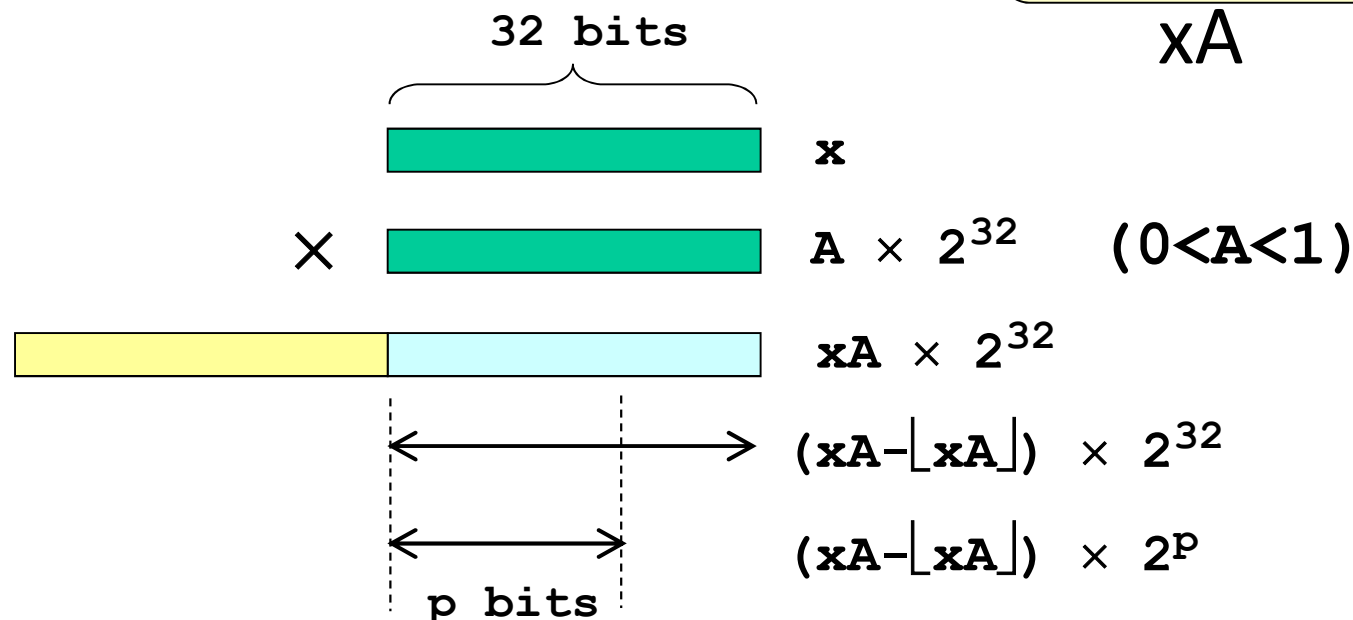
- คัดเลือกเลขโดดบางหลักของคีย์มาพิจารณา
- มั่นใจว่าที่ตัดไปไม่ทำให้เกิดความเอนเอียงในการกระจายของคีย์
- เช่น
 - รหัสนิติศาสตร์ ป.ตรี มีรูปแบบ : xx3xxxxx21
 - ก็ตัดเลข 3 และ 21 ออกจากการพิจารณา
 - $k = 4830109521$,
 - $k1 = \lfloor k / 100 \rfloor$ // $k1 = 48301095$
 - $k2 = \lfloor k1 / 10^6 \rfloor$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

การคูณ (Multiplicative Hashing)

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

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

ส่วนที่เป็น
เศษของ
 xA



การคูณ : Fibonacci Hashing

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

```
size_t multHash(size_t x, size_t p) {  
    size_t s      = 2654435769U;  
    size_t hash = (s * x);  
    return (hash >> (32-p));  
}
```

0.6180339887 ×
2³²

```
for (size_t i = 0; i < 10; i++) {  
    cout << (multHash(i, 10) << ", ");  
}
```

0, 632, 241, 874, 483, 92, 725, 334, 966,

การพับ (Folding)

- แบ่งคีย์ออกเป็นส่วนๆ แล้วนำมา "รวม" กัน
- "รวม" \equiv บวก, xor, ...

2	1	0	2	9	3	8	4	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

44995961 =	10101011101001010101111001
19436921 =	101001010001001010101111001
24473977 =	01011101010111000101111001
44738937 =	1010101010101010100101111001

c++11 std::unordered_map

```
#include <iostream>
#include <unordered_map>

using namespace std;

int main() {
    unordered_map<string, int> facultyCode;
    facultyCode["engineering"] = 21;
    facultyCode["accounting" ] = 26;
    facultyCode["science"    ] = 23;

    cout << facultyCode["engineering"] << endl;
    cout << facultyCode["science"]      << endl;
    cout << facultyCode["communication"] << endl;

    return 0;
}
```


ข้อมูลใด ๆ ก็เปลี่ยนเป็นจำนวนเต็มได้

- float → จำนวนเต็ม

```
int floatToIntBits(float x) {  
    union {  
        float f;  
        int i;  
    } u;  
    u.f = x;  
    return u.i;  
}
```

- สตริง → จำนวนเต็ม

- นำแต่ละตัวอักษรในสตริงมา "รวม" กัน

$$\text{"DATA"} \rightarrow 3 \times 26^3 + 0 \times 26^2 + 19 \times 26^1 + 0 \times 26^0 = 53222$$

- class → จำนวนเต็ม

- แปลงข้อมูลภายในให้เป็นจำนวนเต็มแล้วนำมา "รวม" กัน

c++11 std::hash

```
#include <functional>
using namespace std;
int main () {
    hash<string> hStr;
    hash<float>   hFloat;
    hash<int>     hInt;

    cout << hStr("C++")    << endl; // 2262514926
    cout << hFloat(1.2f)    << endl; // 2462087341
    cout << hInt(123)       << endl; // 123

    return 0;
}
```

```
cout << hash<string>() ("C++") << endl;
cout << hash<float>() (1.2f)   << endl;
cout << hash<int>() (123)      << endl;
```

อยากใช้ Book เป็นคีย์

```
class Book {  
public:  
    string title;  
    int    edition;  
    double price;  
  
    Book(string title, int ed = 1, double price = 199.0) :  
        title(title), edition(ed), price(price)  
    {}  
  
    bool operator==(const Book &rhs) const {  
        return title == rhs.title && edition == rhs.edition;  
    }  
};
```

ต้องมี operator==

ใช้ Book เป็นคีย์ : แบบใช้ hash<Book>

```
namespace std {  
    template<>  
    struct hash<Book> {  
    public:  
        size_t operator() (const Book& b) const {  
            return hash<string>() (b.title) ^  
                hash<int>() (b.edition);  
        }  
    };  
}
```

"รวม" hash ของ title กับ
hash ของ edition

```
unordered_map<Book, string> umap = {  
    { {"Data Structures", 1, 200}, "reserved" },  
    { {"Algorithm", 5, 200}, "available"}  
};  
Book b1("Data Structures", 1);  
Book b2("Data Structures", 3);  
cout << umap[b1] << endl;  
cout << umap[b2] << endl;
```

ใช้ Book เป็นคีย์ : แบบใช้ hash<Book>

```
#include <iostream>
#include <unordered_map>
#include <functional>

using namespace std;
int main() {
    unordered_map<Book, string> umap = {
        { {"Data Structures", 1, 200}, "reserved" },
        { {"Algorithm", 5, 200}, "available"}
    };
    Book b1("Data Structures", 1);
    Book b2("Data Structures", 3);
    Book b3("algorithm", 5);
    cout << umap[b1] << endl;
    cout << umap[b2] << endl;
    cout << umap[b3] << endl;
    cout << (umap[b3] == "") << endl;
    return 0;
}
```

ใช้ Book เป็นคีย์ : แบบใช้ hasher

```
class BookHasher {
public:
    size_t operator() (const Book& b) const {
        return hash<string>() (b.title) ^
            hash<int>() (b.edition);
    }
};
```

```
unordered_map<Book, string, BookHasher> umap = {
    { {"Data Structures", 1, 200}, "reserved" },
    { {"Algorithm", 5, 200}, "available" }
};

Book b1("Data Structures", 1);
Book b2("algorithm", 5);
cout << umap[b1] << endl;
cout << umap[b2] << endl;
```

การ "รวม"

```
size_t hash(char *key) {  
    size_t h = 0;  
    char c;  
    while( (c=*key++) != '\0' ) h = 31*h + c;  
    return h;  
}
```

```
class Point {  
    double x, y;  
};  
...  
size_t hash(Point& p) {  
    size_t h = floatToIntBits(p.x);  
    h ^= 31 * floatToIntBits(p.y);  
    return h;  
}
```

อยากใช้ Book เป็นคีย์

```
class Book {  
public:  
    string title;  
    int    edition;  
    double price;  
  
    Book(string title, int ed = 1, double price = 199.0) :  
        title(title), edition(ed), price(price)  
    {}  
    ...  
};
```


เขียน Hasher class, Equal class

```
class BookHasher {
public:
    size_t operator() (const Book& b) const {
        return hash<string>() (b.title) ^ hash<int>() (b.edition);
    }
};

class BookEqual {
public:
    bool operator() (const Book& b1, const Book b2) const {
        return b1.title==b2.title && b1.edition==b2.edition;
    }
};

unordered_map<Book,string,BookHasher,BookEqual> m;
m[Book("Data Structures", 1, 200)] = "reserved";
m[Book("Algorithm", 5, 200)] = "available";

Book b1("Data Structures", 1);
Book b2("algorithm", 5);

cout << m[b1] << endl;
cout << m[b2] << endl;
```

การแฮชเอกภาพ (Universal Hashing)

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

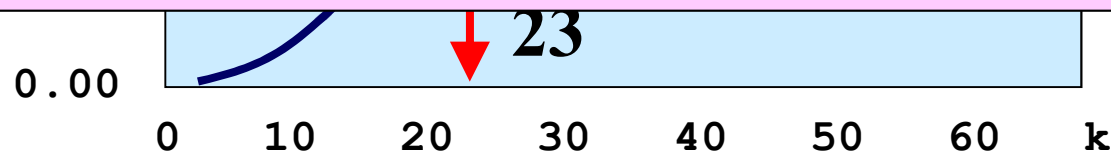
ปฏิกิริยาวันเกิด (Birthday Paradox)

- ต้องมีคนในห้องกี่คนขึ้นไป จึงจะโอกาสเกินครึ่ง
ที่จะมีคนเกิดวันเดือนเดียวกันสองคนขึ้นไป

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

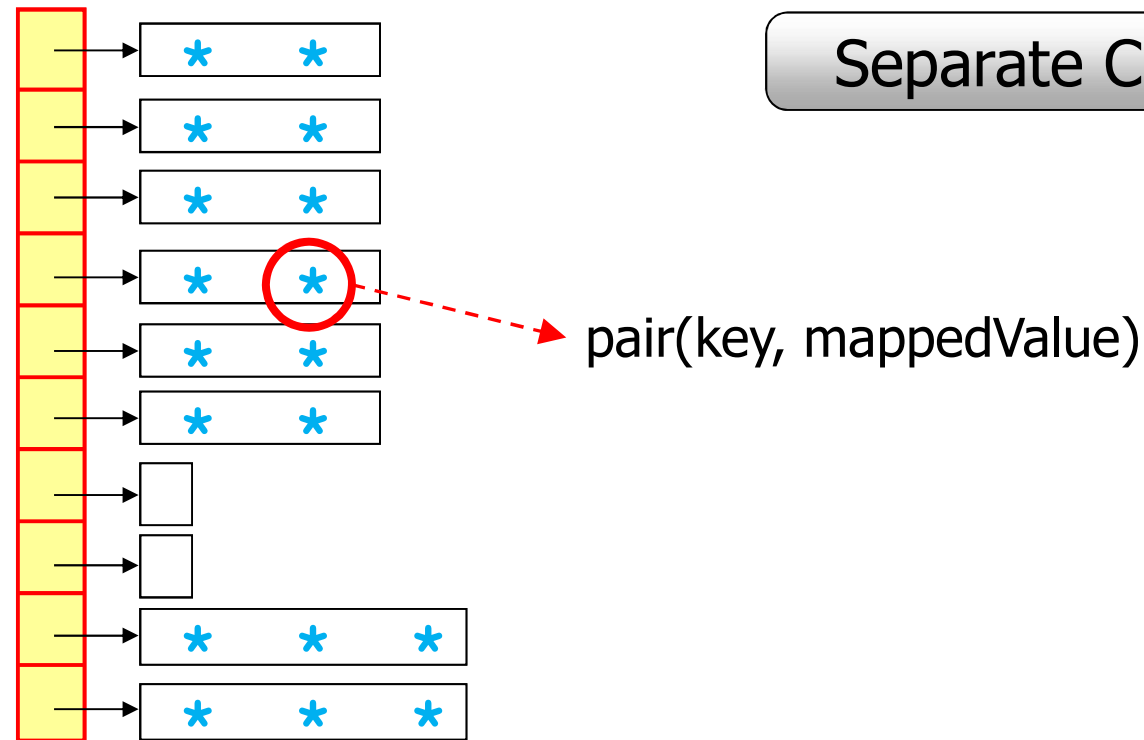
$$1 - \left(\left(\frac{366}{366}\right)\left(\frac{365}{366}\right)\left(\frac{364}{366}\right)\cdots\left(\frac{366-k+1}{366}\right)\right) > 0.5$$

ให้คนคือข้อมูล วันเดือนเกิดแทนหมายเลขช่องในตาราง
มีตารางขนาด 366 ช่อง
แฮชคนเพียง 23 คน ก็มีโอกาสดเกิดการชนเกินครึ่ง



CP::unordered_map<KeyT, MappedT>

Separate Chaining

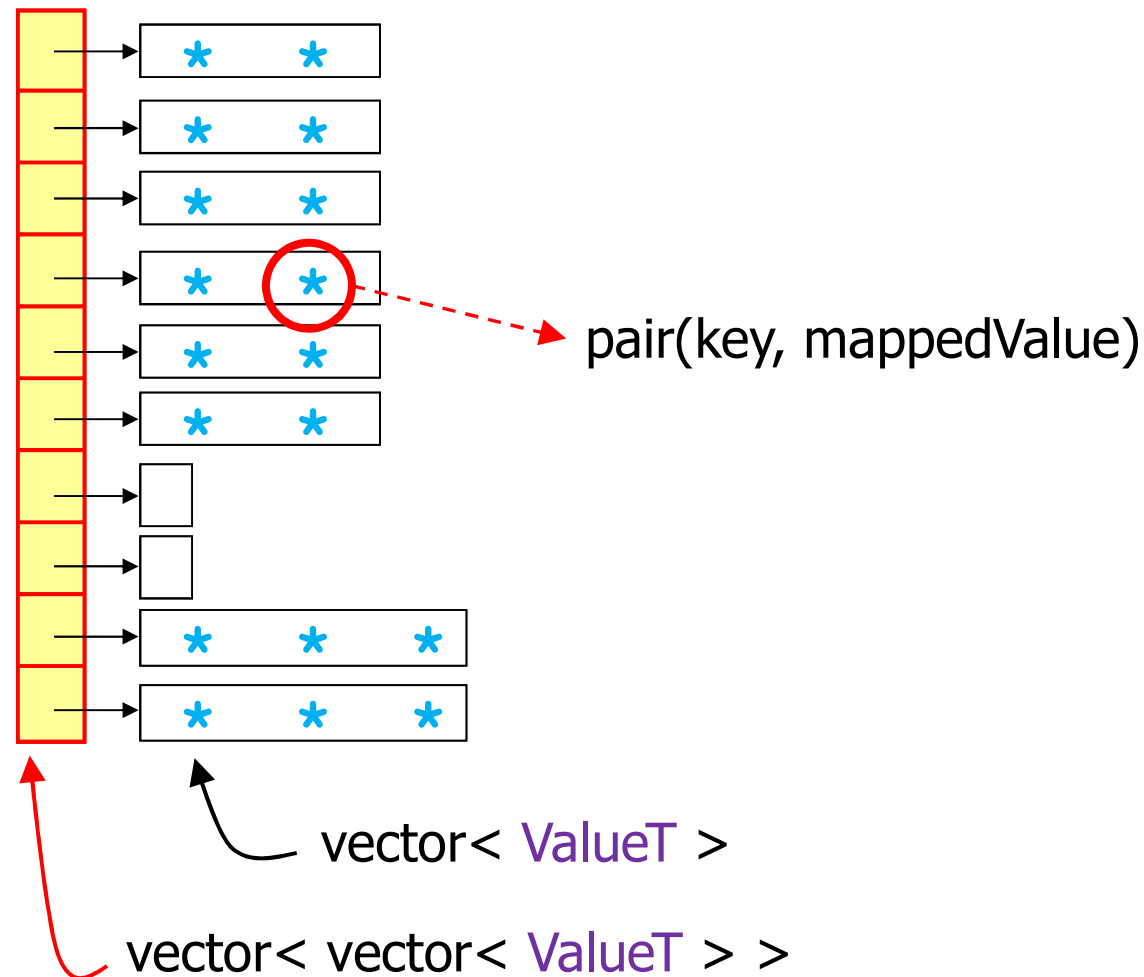


vector< pair<KeyT, MappedT> >

vector< vector< pair<KeyT, MappedT> > >

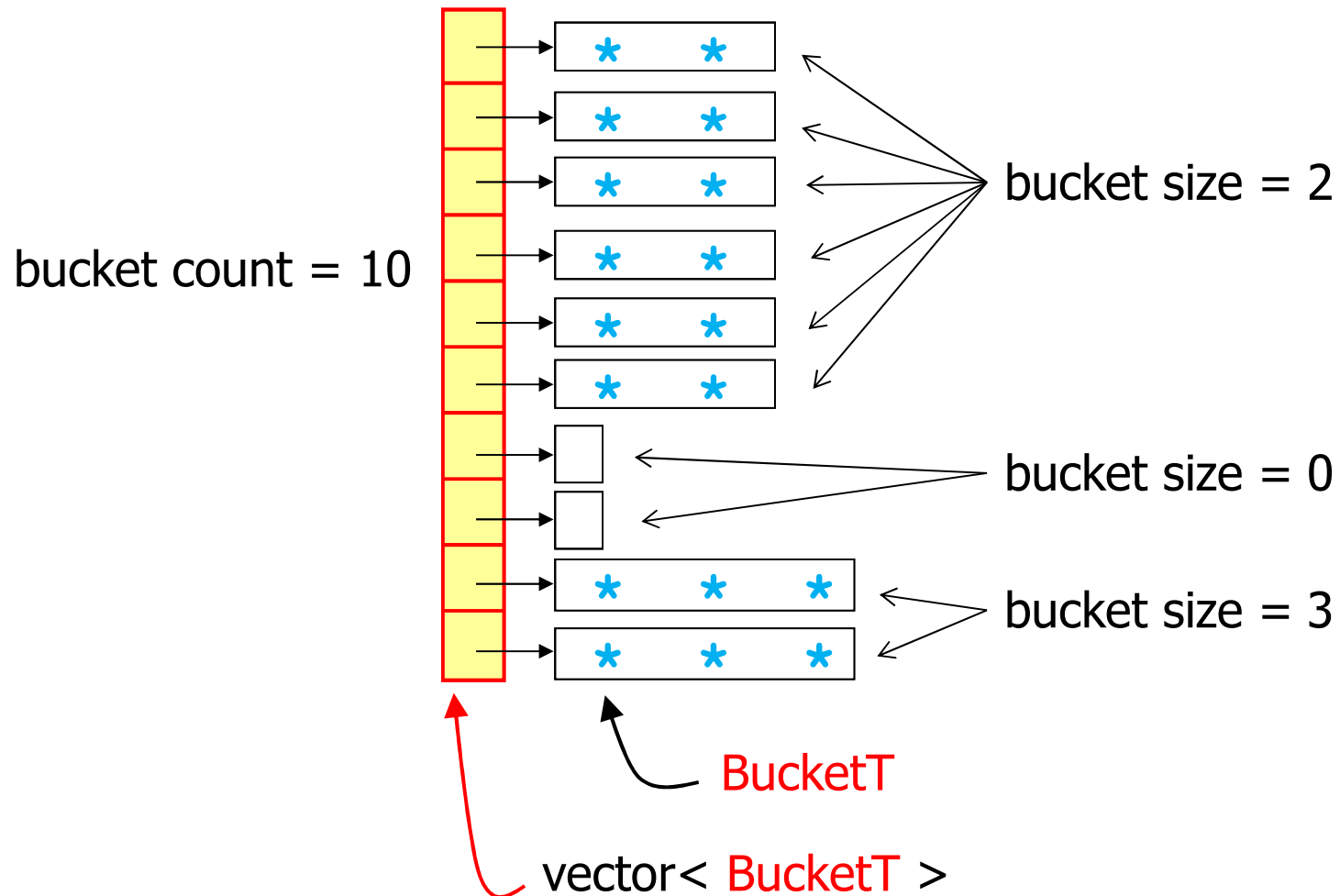
```
typedef pair<KeyT,MappedT> ValueT ;
```

CP::unordered_map<KeyT, MappedT>



```
typedef pair<KeyT,MappedT> ValueT ;  
typedef vector< ValueT > BucketT ;
```

CP::unordered_map<KeyT, MappedT>

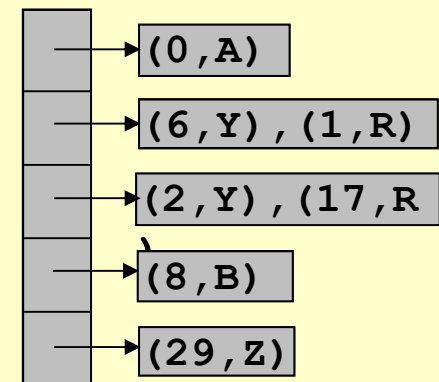


```
typedef pair<KeyT,MappedT> ValueT ;
```

```
typedef vector< ValueT > BucketT ;
```

CP::unordered_map<KeyT, MappedT>

```
template <typename KeyT,  
          typename MappedT,  
          typename HasherT = std::hash<KeyT>,  
          typename EqualT = std::equal_to<KeyT> >  
class unordered_map {  
protected:  
    typedef std::pair<KeyT, MappedT>    ValueT;  
    typedef std::vector<ValueT>         BucketT;  
    ...  
  
    std::vector<BucketT> mBuckets;  
    size_t               mSize;  
    HasherT              mHasher;  
    EqualT               mEqual;  
    float                mMaxLoadFactor;
```



ใช้ในการเปรียบเทียบ ระหว่างการค้น key
hash function เพื่อคำนวณตำแหน่ง bucket

default constructor

```
template <typename KeyT,  
          typename MappedT,  
          typename HasherT = std::hash<KeyT>,  
          typename EqualT  = std::equal_to<KeyT> >  
class unordered_map {  
    ...  
    std::vector<BucketT> mBuckets;  
    size_t               mSize;  
    HasherT              mHasher;  
    EqualT               mEqual;  
    float                mMaxLoadFactor;  
    ...  
    unordered_map() :  
        mBuckets( std::vector<BucketT>(11) ), mSize(0),  
        mHasher( HasherT() ), mEqual( EqualT() ),  
        mMaxLoadFactor(1.0)  
    { }  
};
```


copy constructor

```
template <typename KeyT,  
          typename MappedT,  
          typename HasherT = std::hash<KeyT>,  
          typename EqualT   = std::equal_to<KeyT> >  
class unordered_map {  
    ...  
    std::vector<BucketT> mBuckets;  
    size_t               mSize;  
    HasherT              mHasher;  
    EqualT               mEqual;  
    float                mMaxLoadFactor;  
    ...  
    unordered_map(const  
        unordered_map<KeyT, MappedT, HasherT, EqualT> & other) :  
        mBuckets(other.mBuckets), mSize(other.mSize),  
        mHasher(other.mHasher), mEqual(other.mEqual),  
        mMaxLoadFactor(other.mMaxLoadFactor)  
    { }  
};
```

copy assignment

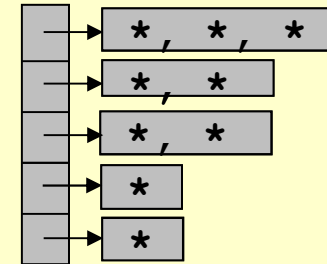
```
class unordered_map {
    ...
    std::vector<BucketT> mBuckets;
    size_t                mSize;
    HasherT               mHasher;
    EqualT                mEqual;
    float                 mMaxLoadFactor;
    ...
    unordered_map<KeyT, MappedT, HasherT, EqualT>&
        operator=(unordered_map<KeyT, MappedT, HasherT, EqualT>
                    other) {
        using std::swap;
        swap(this->mBuckets,      other.mBuckets);
        swap(this->mSize,          other.mSize    );
        swap(this->mHasher,        other.mHasher  );
        swap(this->mEqual,         other.mEqual   );
        swap(this->mMaxLoadFactor, other.mMaxLoadFactor);
        return *this;
    }
}
```

CP::unordered_map<KeyT, MappedT>

```
template < ... >
class unordered_map {
public:
    bool      empty()                {...}
    size_t    size()                 {...}
    size_t    bucket_count()         {...}
    size_t    bucket_size(size_t n) {...}
    float     load_factor()          {...}
    float     max_load_factor()      {...}
    void      max_load_factor(float z) {...}

    iterator  begin()                {...}
    iterator  end()                  {...}

    MappedT& operator[] (const KeyT& key) {...}
    void      clear()                {...}
    void      rehash(size_t n)       {...}
    size_t    erase(const KeyT &key) {...}
    pair<iterator, bool> insert(const ValueT& val) {...}
```

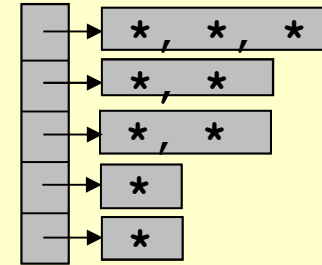


$$\lambda = 9/5 = 1.8$$

```
m["ok"] = 27;
cout << m["ok"];
```

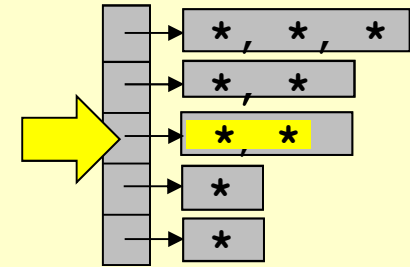
CP::unordered_map<KeyT, MappedT>

```
class unordered_map {  
    ...  
    std::vector<BucketT> mBuckets;  
    size_t                mSize;  
    float                 mMaxLoadFactor  
    ...  
    bool    empty() { return mSize == 0; }  
    size_t  size()  { return mSize; }  
    size_t  bucket_count() {  
        return mBuckets.size();  
    }  
    size_t  bucket_size(size_t n) {  
        return mBuckets[n].size();  
    }  
    float   load_factor() {  
        return (float)mSize/mBuckets.size();  
    }  
    float   max_load_factor() {  
        return mMaxLoadFactor;  
    }  
};
```



unordered_map: operator[]

```
size_t hash_to_bucket(const KeyT& key) {  
    return mHasher(key) % mBuckets.size();  
}
```



ValueIterator

```
find_in_bucket(BucketT& b, const KeyT& key) {  
    for(ValueIterator it = b.begin(); it != b.end(); it++){  
        if (mEqual(it->first, key)) return it;  
    }  
    return b.end();  
}
```

```
MappedT& operator[] (const KeyT& key) {  
    size_t      bi = hash_to_bucket(key);  
    ValueIterator it = find_in_bucket(mBuckets[bi], key);  
    // ถ้าหาไม่พบ ต้องเพิ่ม pair(key, ค่า default ของ mapped value)  
    return it->second;  
}
```

operator[] หาไม่พบเพิ่มตัวคู่ใหม่

ValueIterator

```
insert_to_bucket(const ValueT& val, size_t& bi) {  
    if ( ตารางแน่นเกินไป ) { ปรับตาราง }  
    ++mSize;  
    return mBuckets[bi].insert(mBuckets[bi].end(), val);  
}
```

ผลการ insert ใน vector คือ
iterator ไปยังข้อมูลใหม่ที่ถูกเพิ่ม

เพิ่ม val ด้านท้าย

```
MappedT& operator[] (const KeyT& key) {  
    size_t      bi = hash_to_bucket(key);  
    ValueIterator it = find_in_bucket(mBuckets[bi], key);  
  
    if (it == mBuckets[bi].end()) {  
        it = insert_to_bucket(make_pair(key, MappedT()), bi);  
    }  
  
    return it->second;  
}
```

unordered_map: erase

```
size_t erase(const KeyT & key) {  
    size_t      bi = hash_to_bucket(key);  
    ValueIterator it = find_in_bucket(mBuckets[bi], key);  
    if (it == mBuckets[bi].end()) {  
        return 0;  // erase 0 element  
    } else {  
        mBuckets[bi].erase(it);  
        mSize--;  
        return 1;  // erase 1 element  
    }  
}
```

ผลของการ **erase** คือ จำนวนข้อมูลที่ถูกลบ

- 0 คือ ไม่พบ key ไม่มีการลบเกิดขึ้น
- 1 คือ พบ key มีการลบ pair ของ key และ mapped value ของ key นั้น

unordered_map: insert

```
pair<iterator,bool> insert(const ValueT& val) {  
    pair<iterator,bool> result;  
    const KeyT& key = val.first;  
    size_t      bi = hash_to_bucket(key);  
    ValueIterator it = find_in_bucket(mBuckets[bi], key);  
    result.second = false;  
    if (it == mBuckets[bi].end()) {  
        it = insert_to_bucket(val, bi);  
        result.second = true;  
    }  
    result.first = iterator(it,  
                             mBuckets.begin()+bi,  
                             mBuckets.end());  
    return result;  
}
```

bi อาจเปลี่ยน
ถ้ามีการปรับตาราง

iterator ของ
unordered_map

```
ValueIterator insert_to_bucket(const ValueT& val, size_t& bi) {  
    if ( ตารางแน่นเกินไป ) { ปรับตาราง }  
    ++mSize;  
    return mBuckets[bi].insert(mBuckets[bi].end(), val);  
}
```

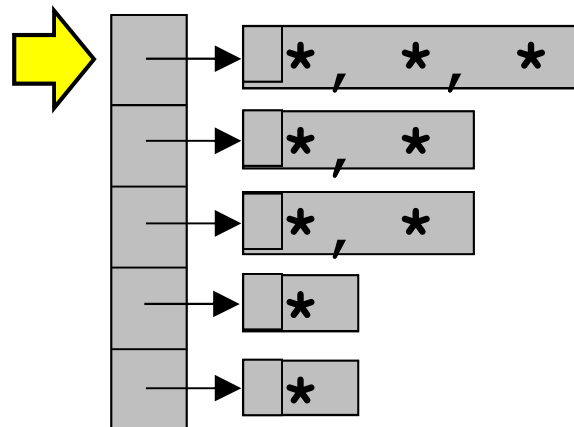

unordered_map: clear

```
void clear() {  
    for (vector<BucketT>::iterator it = mBuckets.begin();  
        it != mBuckets.end();  
        ++it) {  
        (*it).clear();  
    }  
    mSize = 0;  
}
```

```
void clear() {  
    for (: auto & bucket : mBuckets) {  
        bucket.clear();  
    }  
    mSize = 0;  
}
```

for each bucket in mBuckets

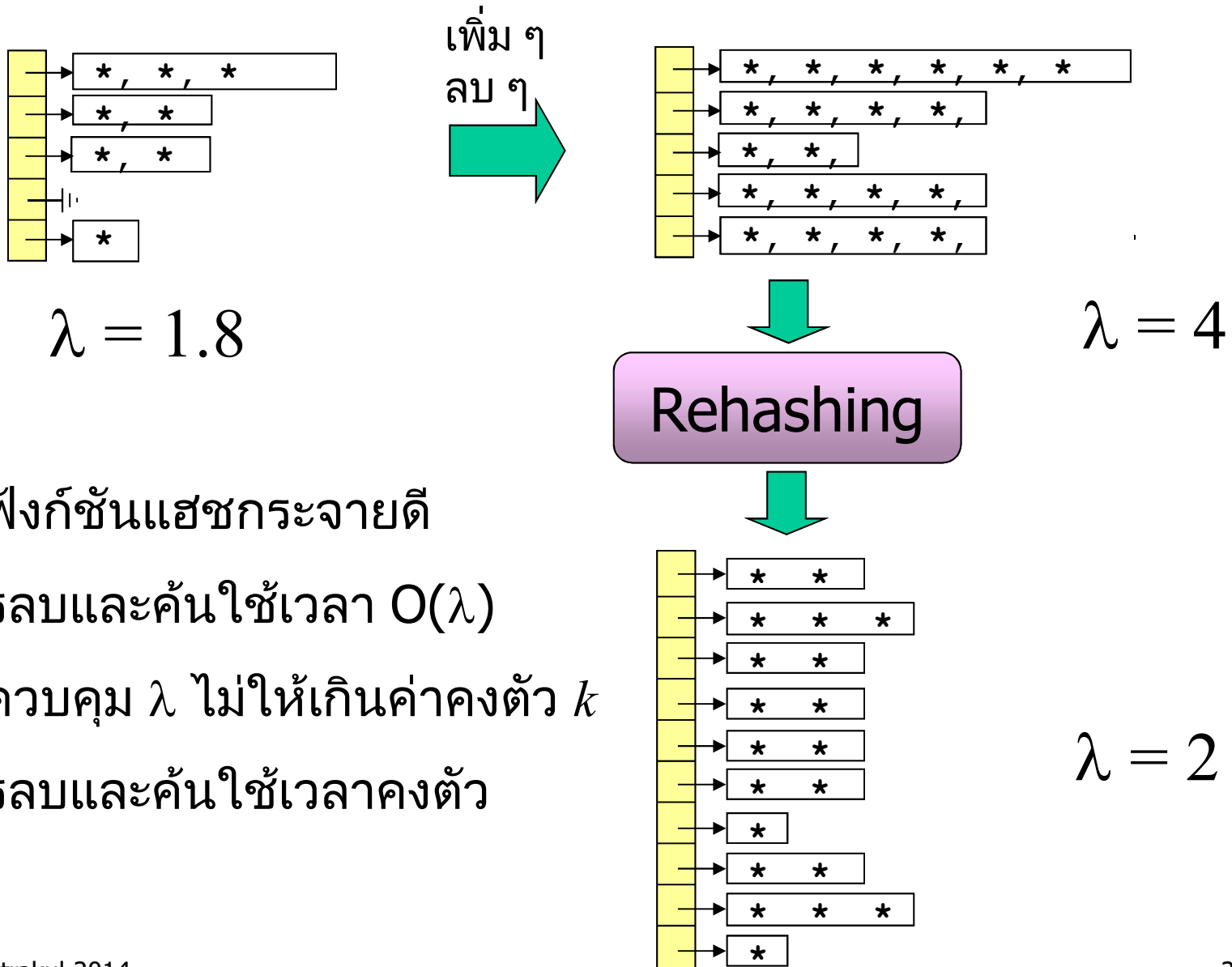
vector<BucketT> mBuckets



unordered_map: destructor

```
class unordered_map {  
    ...  
    ~unordered_map() {  
        clear();  
    }  
    ...  
    void clear() {  
        for ( auto& bucket : mBuckets ) {  
            bucket.clear();  
        }  
        mSize = 0;  
    }  
    ...  
}
```

Rehashing



ถ้าฟังก์ชันแฮชกระจายดี
การลบและค้นใช้เวลา $O(\lambda)$
ถ้าควบคุม λ ไม่ให้เกินค่าคงตัว k
การลบและค้นใช้เวลาคงตัว

ถ้า "แน่น" เกิน ต้อง rehash

```
void rehash(size_t m) {  
    if ( m <= mBuckets.size() &&  
        load_factor() <= max_load_factor() ) return;  
    m = std::max(m, (size_t)(0.5+mSize/mMaxLoadFactor));  
    m = *std::lower_bound(PRIMES, PRIMES+N_PRIMES, m);  
    vector<ValueT> tmp;  
    for (auto& val : *this) tmp.push_back(val);  
    this->clear();  
    mBuckets.resize(m);  
    for (auto& val : tmp ) this->insert(val);  
}
```

$$m_{max} = \frac{n m}{m_{max}}$$

```
ValueIterator insert_to_bucket(const ValueT& val, size_t& bi) {  
    if (load_factor() > max_load_factor()) {  
        rehash(2*bucket_count());  
        bi = hash_to_bucket(val.first);  
    }  
    ++mSize;  
    return mBuckets[bi].insert(mBuckets[bi].end(), val);  
}
```

ถ้ามีการปรับตาราง
ต้องเปลี่ยนค่า **bi**

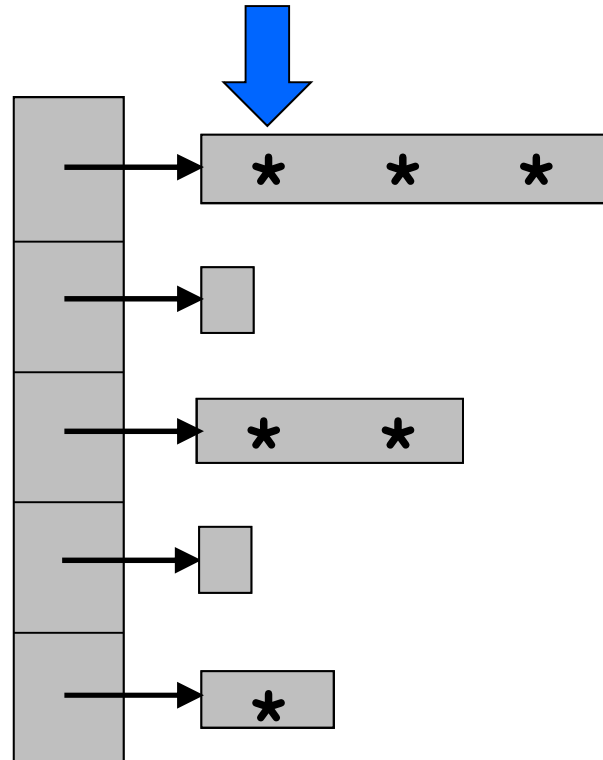
ขอปรับขนาดตารางเป็นจำนวนเฉพาะ

```
const size_t      N_PRIMES      = 256;
const unsigned long PRIMES[256] = {
    2ul, 3ul, 5ul, 7ul, 11ul, 13ul, 17ul, 19ul, 23ul, 29ul,
    ...
};
```

คืนตำแหน่งแรกที่พบในช่วง [**PRIMES**, **PRIMES+N_PRIMES**) ที่มีค่าไม่น้อยกว่า **m**

```
void rehash(size_t m) {
    if ( n <= mBuckets.size() &&
         load_factor() <= max_load_factor() ) return;
    m = std::max(m, (size_t)(0.5+mSize/mMaxLoadFactor));
    m = *std::lower_bound(PRIMES, PRIMES+N_PRIMES, m);
    vector<ValueT> tmp;
    for (auto& val : *this) tmp.push_back(val);
    this->clear();
    mBuckets.resize(m);
    for (auto& val : tmp ) this->insert(val);
}
```

unordered<KeyT,MappedT>::iterator



unordered<KeyT,MappedT>::iterator

```
class unordered_map {  
protected
```

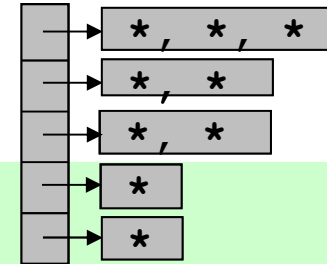
```
...
```

```
class hashtable_iterator {
```

```
...
```

```
public:
```

```
    hashtable_iterator()                {...}  
    hashtable_iterator& operator++()    {...} // ++it  
    hashtable_iterator operator++(int) {...} // it++  
    ValueT & operator*()                {...} // *it  
    ValueT * operator->()                {...} // it->first  
    bool operator!=(const hashtable_iterator &other) {...}  
    bool operator==(const hashtable_iterator &other) {...}  
};
```



```
public:
```

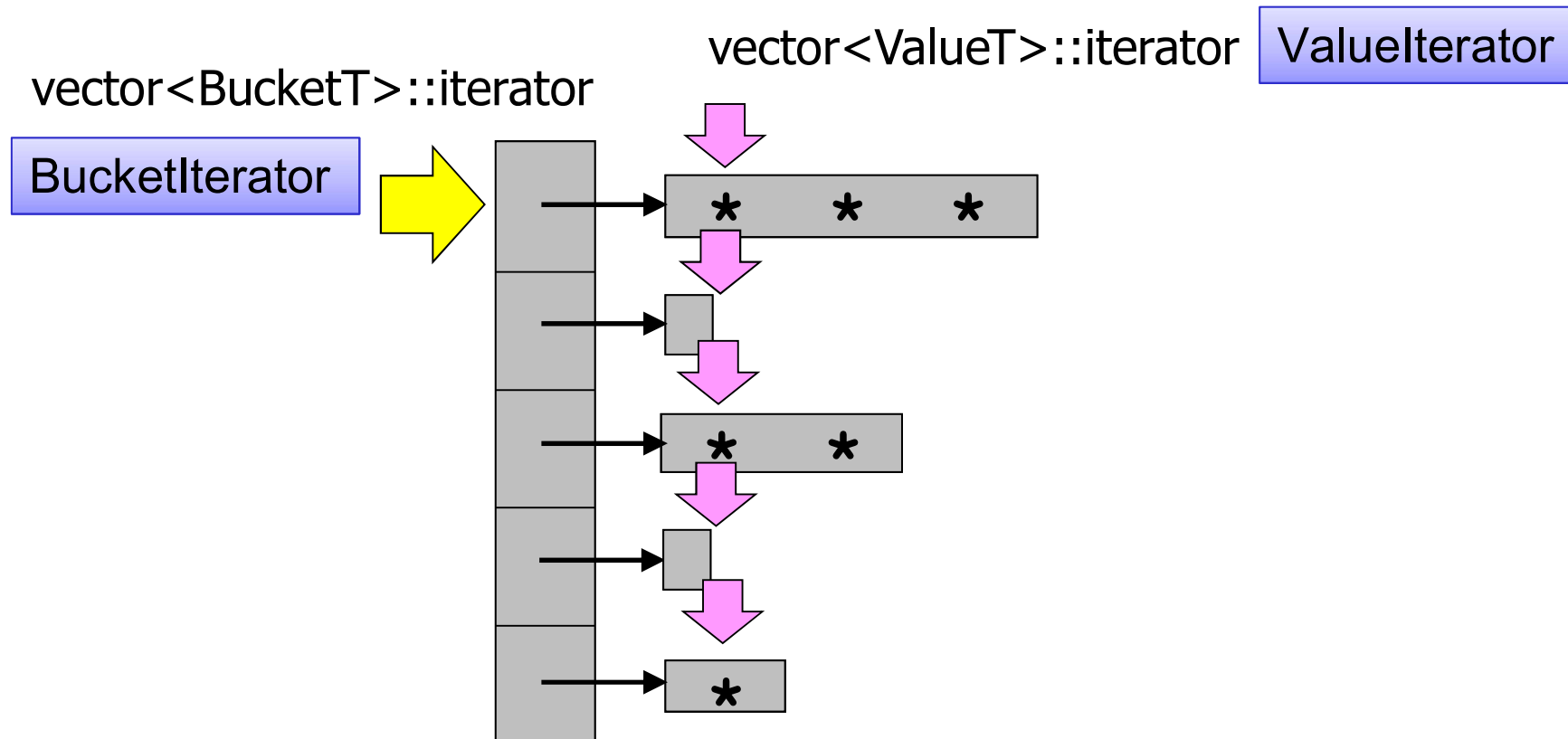
```
typedef hashtable_iterator iterator;
```

```
...
```

```
}
```

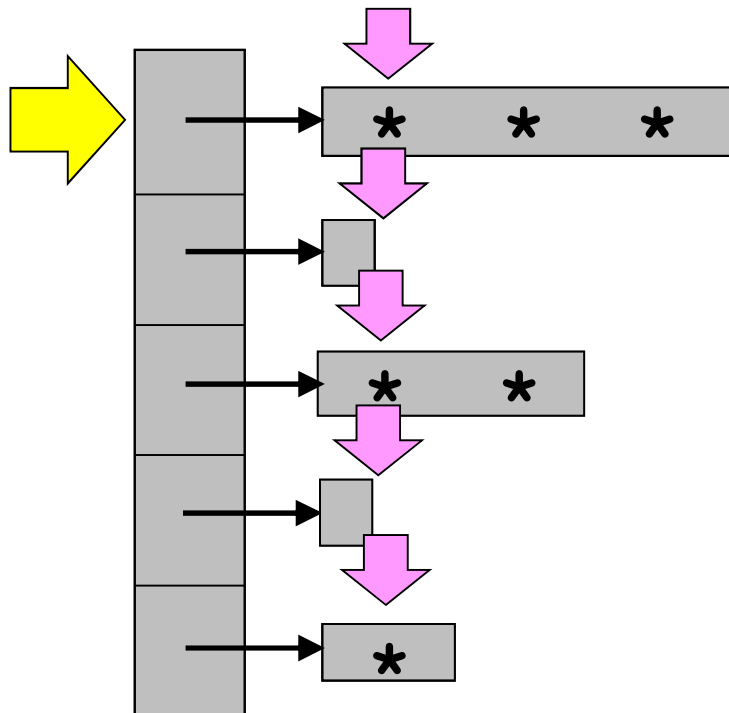
```
unordered_map<string,int>::iterator it = m.begin();
```

++iterator


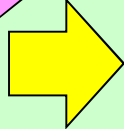
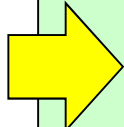



```
class unordered_map {  
    public:  
        std::vector<BucketT> mBuckets;  
        size_t                mSize;  
        ...  
};
```


++iterator



```
it = m.begin();  
++it;  
...
```

```
class hashtable_iterator {  
protected:  
    ValueIterator mCurValueIter;   
    BucketIterator mCurBucketIter;   
  
    void to_next_data( ) {  
        while (mCurBucketIter != mBuckets.end() &&  
               mCurValueIter == mCurBucketIter->end()) {  
 mCurBucketIter++;  
            if (mCurBucketIter == mBuckets.end()) break;  
            mCurValueIter = mCurBucketIter->begin();  
        }  
    }  
public:  
    hashtable_iterator& operator++( ) {  
 mCurValueIter++;  
        to_next_data();  
        return (*this);  
    }  
}
```

inner class ใช้ outer's fields ไม่ได้

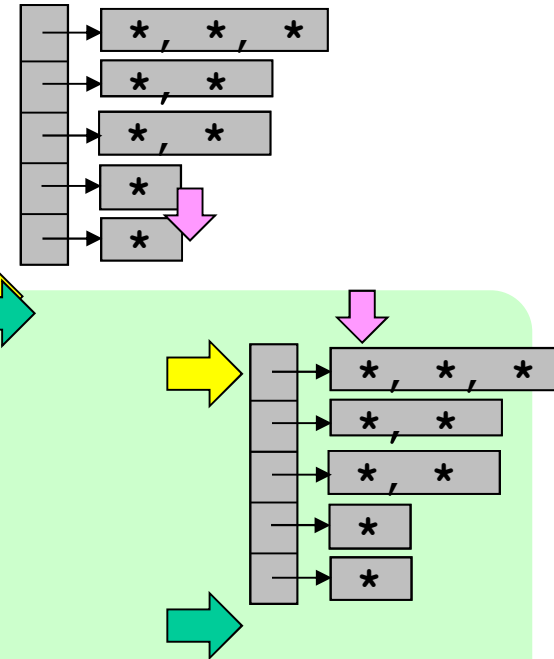
```
class unordered_map {  
protected  
    vector<BucketT> mBuckets;  
    size_t          mSize;  
    ...
```

```
class hashtable_iterator {  
    ValueIterator mCurValueItr;  
    BucketIterator mCurBucketItr;  
    BucketIterator mEndBucketItr;  
    ...
```

```
void to_next_data( ) {  
    while ( mCurBucketItr != mBuckets.end() &&  
            mCurValueItr == mCurBucketItr->end() ) {  
        mCurBucketItr++;  
        if (mCurBucketItr == mBuckets.end()) break;  
        mCurValueItr = mCurBucketItr->begin();  
    }  
}
```

ให้ `mEndBucketItr` เก็บ `mBuckets.end()`
ตอนสร้าง iterator

ทำไม่ได้



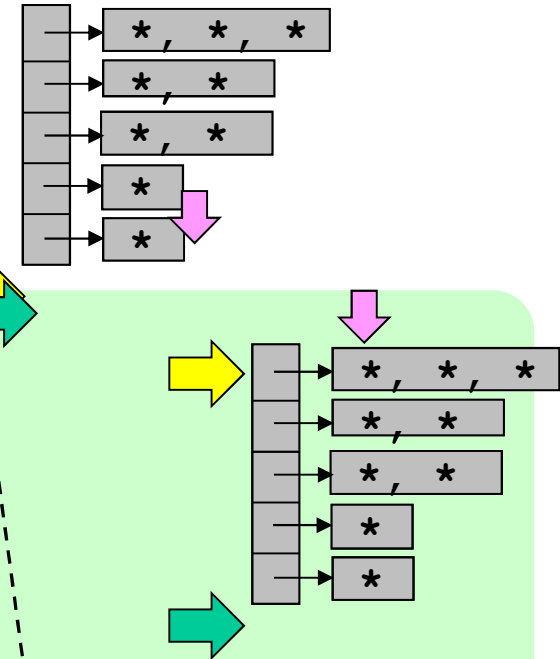
iterator ต้องจำ mBuckets.end() ด้วย

```
class unordered_map {  
protected  
    vector<BucketT> mBuckets;  
    size_t           mSize;  
  
    ...
```

```
class hashtable_iterator {  
    ValueIterator mCurValueItr;  
    BucketIterator mCurBucketItr;  
    BucketIterator mEndBucketItr;  
    ...
```

```
void to_next_data( ) {  
    while ( mCurBucketItr != mEndBucketItr &&  
            mCurValueItr == mCurBucketItr->end() ) {  
        mCurBucketItr++;  
        if (mCurBucketItr == mEndBucketItr) break;  
        mCurValueItr = mCurBucketItr->begin();  
    }  
}
```

ให้ **mEndBucketItr** เก็บ **mBuckets.end()**
ตอนสร้าง iterator



`++it` ก็ได้, `it++` ก็ได้ แต่...

```
class hashtable_iterator {  
    ValueIterator mCurValueItr;  
    BucketIterator mCurBucketItr;  
    BucketIterator mEndBucketItr;  
    ...
```

```
public:
```

```
    hashtable_iterator& operator++() { // ++it
```

```
        mCurValueItr++;
```

```
        to_next_data();
```

```
        return (*this);
```

```
    }
```

```
    hashtable_iterator operator++(int) { // it++
```

```
        hashtable_iterator tmp(*this);
```

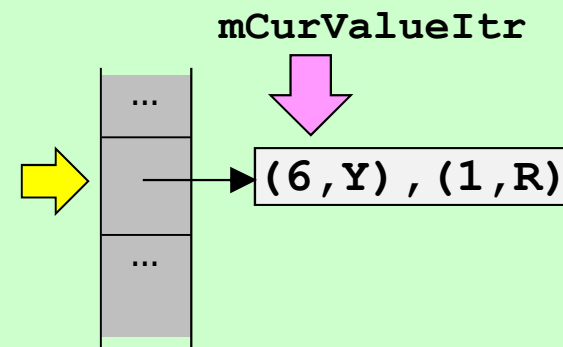
```
        operator++();
```

```
        return tmp;
```

```
    }
```

```
    ...
```

```
};
```

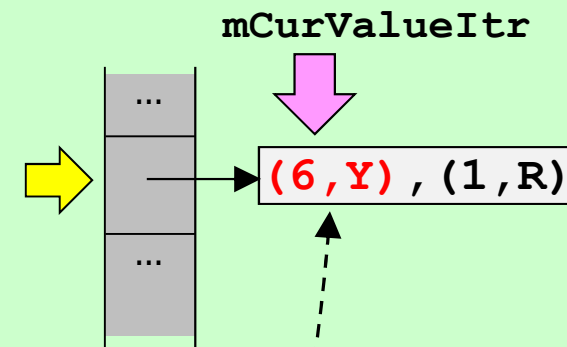


`++(++it)` it เลื่อนสองครั้ง

`(it++)++` it เลื่อนครั้งเดียว!

*it กับ it->

```
class hashtable_iterator {  
    ValueIterator mCurValueItr;  
    BucketIterator mCurBucketItr;  
    BucketIterator mEndBucketItr;  
    ...  
public:  
    typedef ValueT & reference;  
    typedef ValueT * pointer;  
  
    reference operator*() {  
        return *mCurValueItr;  
    }  
  
    pointer operator->() {  
        return &(*mCurValueItr);  
    }  
    ...  
};
```



`*mCurValueItr`
ก็คือ pair `(6, Y)`

```
it = m.begin();  
cout << (*it).first;
```

```
it = m.begin();  
cout << it->first;
```

`it1 == it2` กับ `it1 != it2`

```
class hashtable_iterator {
```

```
protected:
```

```
    ValueIterator mCurValueItr;
```

```
    BucketIterator mCurBucketItr;
```

```
    BucketIterator mEndBucketItr;
```

```
public:
```

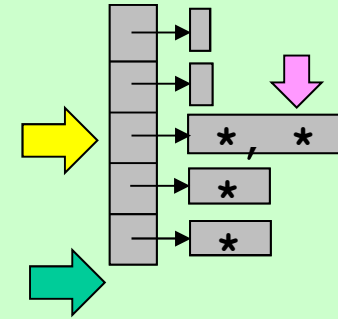
```
    ...
```

```
    bool operator==(const hashtable_iterator &other) {  
        return mCurValueItr == other.mCurValueItr;  
    }
```

```
    bool operator!=(const hashtable_iterator &other) {  
        return mCurValueItr != other.mCurValueItr;  
    }
```

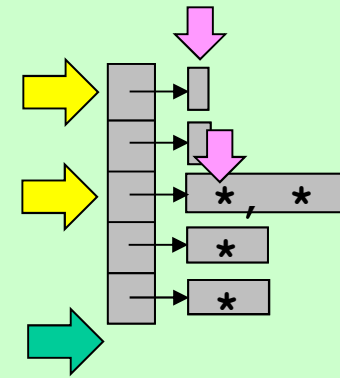
```
    ...
```

```
};
```



hashtable_iterator :: constructor

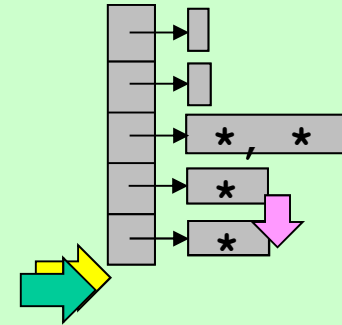
```
class hashtable_iterator {  
protected:  
    ValueIterator    mCurValueItr;  
    BucketIterator   mCurBucketItr;  
    BucketIterator   mEndBucketItr;  
public:  
    hashtable_iterator(ValueIterator  valueItr,  
                       BucketIterator bucketItr,  
                       BucketIterator endBucketItr) :  
        mCurValueItr(valueItr),  
        mCurBucketItr(bucketItr),  
        mEndBucketItr(endBucketItr)  
    {  
        to_next_data();  
    }  
    ...  
};
```



```
iterator begin() {  
    return iterator( mBuckets.begin()->begin(),  
                    mBuckets.begin(),  
                    mBuckets.end() );  
}
```

unordered_map: end()

```
class hashtable_iterator {  
protected:  
    ValueIterator    mCurValueItr;  
    BucketIterator    mCurBucketItr;  
    BucketIterator    mEndBucketItr;  
public:  
    hashtable_iterator(ValueIterator    valueItr,  
                        BucketIterator    bucketItr,  
                        BucketIterator    endBucketItr) :  
        mCurValueItr(valueItr),  
        mCurBucketItr(bucketItr),  
        mEndBucketItr(endBucketItr)  
    {  
        to_next_data();  
    }  
};
```



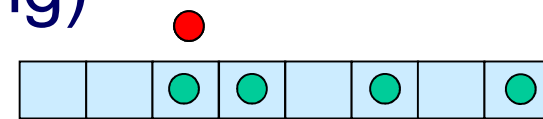
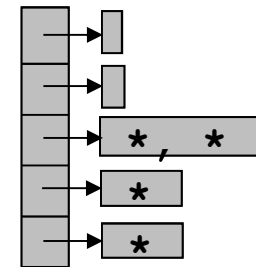
```
    iterator end() {  
        return iterator( mBuckets[mBuckets.size()-1].end(),  
                          mBuckets.end(),  
                          mBuckets.end() );  
    }
```


อย่าลืม default constructor

```
class unordered_map {  
    ...  
    class hashtable_iterator {  
        ...  
        hashtable_iterator() { }  
        ...  
    };  
    ...  
    pair<iterator,bool> insert(const ValueT& val) {  
        pair<iterator,bool> result;  
        ...  
        return result;  
    }  
    ...  
}
```

การแก้ปัญหาคาการชนแบบอื่น

- แบบแยกกันโยง (separate chaining)
 - แต่ละช่องในตารางเก็บรายการโยงของข้อมูล
 - ข้อมูลที่ชนกันเก็บอยู่ด้วยกัน ไม่กระทบข้อมูลอื่น
- 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)

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

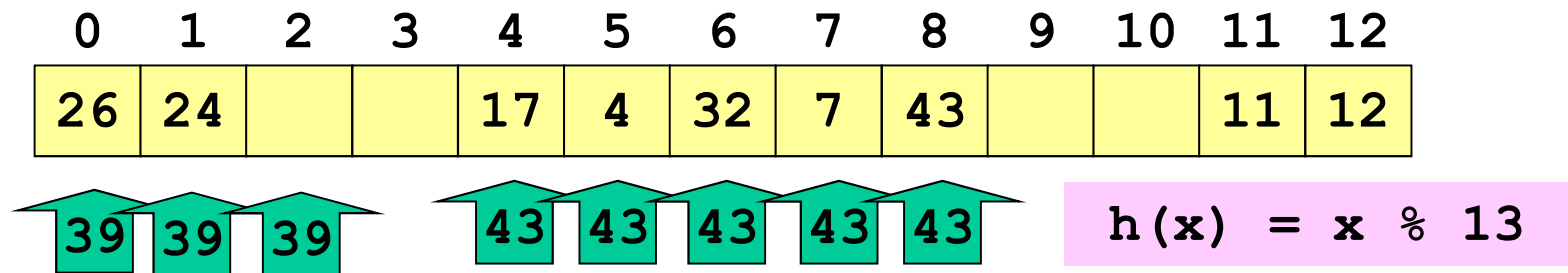
$$h_j(x) = (h_{j-1}(x) + 1) \% m$$

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

ใช้ $h(x) = x \% 13$ แล้วเพิ่มข้อมูลที่มีคีย์ตามลำดับดังนี้

17 32 26 7 4 43 12 11 24

Linear Probing : คำน



หาไม่พบ เมื่อพบช่องว่าง

Linear Probing : ลบ

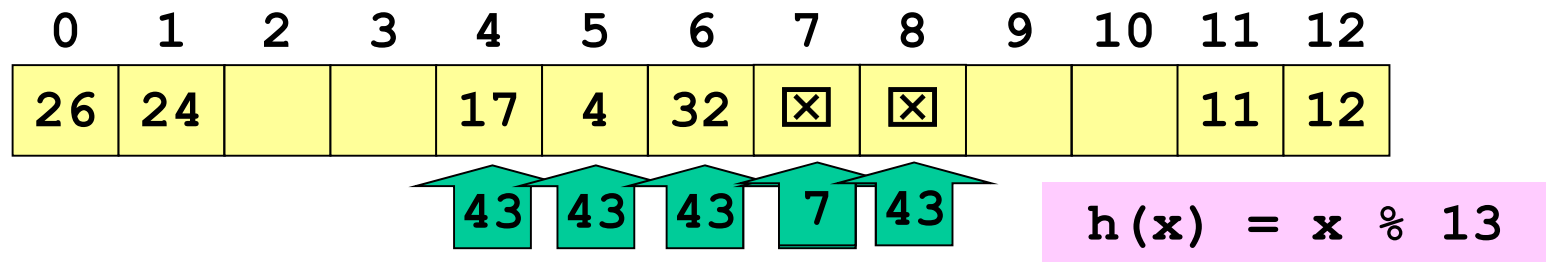
0	1	2	3	4	5	6	7	8	9	10	11	12
26	24			17	4	32	7	43			11	12

43	43	43	7
----	----	----	---

$h(x) = x \% 13$

ไม่พบ 43 เพราะจบการค้นที่ช่องว่าง
ทั้งๆ ที่มี 43 อยู่

Linear Probing : ลป



สถานะของช่องเก็บข้อมูล (bucket)

- แต่ละช่องมี 3 สถานะ
 - 0 : empty : ช่องว่าง ๆ ไม่เคยมีข้อมูลมาเก็บเลย
 - 1 : deleted : ช่องที่เก็บข้อมูลที่ถูกลบไปแล้ว
 - 2 : data : ช่องที่มีข้อมูลเก็บอยู่

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

ข้อมูลที่เก็บในตาราง

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

```

template <...>
class unordered_map {
protected:
    typedef pair<KeyT, MappedT> ValueT;

    class BucketT {
    public:
        ValueT value;
        unsigned char status;

        bool available()    { return status < 2; }
        bool empty()        { return status == 0; }
        bool has_data()     { return status == 2; }
        void mark_deleted() { status = 1; }
        void mark_empty()   { status = 0; }
        void mark_data()    { status = 2; }


    };

    vector<BucketT> mBuckets;
    
```

0 = empty, 1 = deleted, 2 = data

การเปลี่ยนสถานะของ bucket

- constructor → empty
- m.insert(val) → mark_data
- m["X"] = 2 → mark_data
- m.erase("X") → mark_deleted
- m.clear() → mark_empty
- m.rehash(...)
 - clear → mark_empty
 - insert → mark_data

	A		Y		Q	
--	---	---	---	--	---	--

มาดู iterator กันก่อน

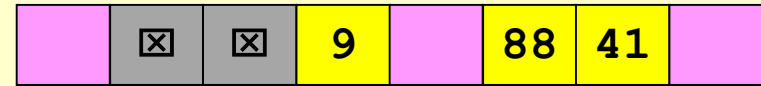
```
template <...>
class unordered_map {
protected:
    typedef pair<KeyT,MappedT> ValueT;
    class BucketT {...}
    vector<BucketT> mBuckets;

    class hashtable_iterator {
    protected:

    public:
        ...
    }

public:
    typedef hashtable_iterator iterator;
    iterator begin() {...}
    iterator end()   {...}
    ...
}
```

mBuckets



```

class unordered_map {
    ...
    typedef typename vector<BucketT>::iterator BucketIterator;
    class hashtable_iterator {
    protected:
        BucketIterator mCurBucketItr;
        BucketIterator mEndBucketItr;
        void to_next_data() {
            while ( mCurBucketItr != mEndBucketItr &&
                    !mCurBucketItr->has_data() ) {
                mCurBucketItr++;
            }
        }
    public:
        hashtable_iterator(BucketIterator bucket,
                           BucketIterator endBucket) :
            mCurBucketItr(bucket), mEndBucketItr(endBucket) {
            to_next_data();
        }
    }
    public:
        iterator begin() {
            return iterator( mBuckets.begin(), mBuckets.end() );
        }

```

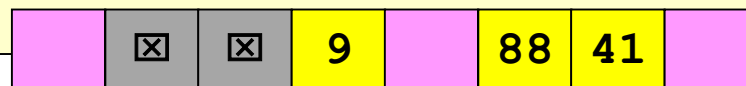
mBuckets



++it กับ it++

```
class hashtable_iterator {  
protected:  
    BucketIterator mCurBucketItr;  
    BucketIterator mEndBucketItr;  
    void to_next_data() {...}  
  
public:  
    ...  
    hashtable_iterator& operator++() { // ++it  
        mCurBucketItr++;  
        to_next_data();  
        return (*this);  
    }  
    hashtable_iterator operator++(int) { // it++  
        hashtable_iterator tmp(*this);  
        operator++();  
        return tmp;  
    }  
}
```

mBuckets



*it กับ it->

```
class hashtable_iterator {  
protected:  
    BucketIterator mCurBucketItr;  
    BucketIterator mEndBucketItr;  
    void to_next_data() {...}
```

public:

...

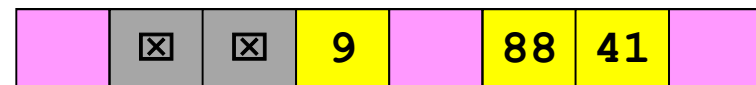
```
ValueT & operator*() {  
    return mCurBucketItr->value;  
}
```

```
it = m.begin();  
(*it).second = 78;
```

```
ValueT * operator->() {  
    return &(mCurBucketItr->value);  
}
```

```
it = m.begin();  
it->second = 78;
```

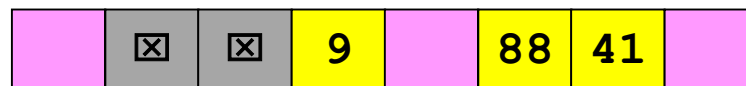
```
class BucketT {  
    ValueT value;  
    unsigned char status;  
};
```



== กับ !=

```
class hashtable_iterator {  
protected:  
    BucketIterator mCurBucketItr;  
    BucketIterator mEndBucketItr;  
    void to_next_data() {...}  
  
public:  
    ...  
    bool operator!=(const hashtable_iterator &other) {  
        return (mCurBucketItr != other.mCurBucketItr);  
    }  
  
    bool operator==(const hashtable_iterator &other) {  
        return (mCurBucketItr == other.mCurBucketItr);  
    }  
}
```

mBuckets




unordered_map (linear probing)

```
template <...>
class unordered_map {
    protected:
        typedef pair<KeyT, MappedT> ValueT;
        class BucketT {...}
        class hashtable_iterator {...}

        vector<BucketT> mBuckets;
        size_t          mSize;
        HasherT          mHasher;           // ใช้ใน hash_to_bucket
        EqualT           mEqual;            // ใช้ใน find_position
        float            mMaxLoadFactor;    // ใช้ใน insert_to_position
        size_t           mUsed;             // # data + # deleted

        size_t hash_to_bucket(const KeyT& key) {
            return mHasher(key) % mBuckets.size();
        }
        size_t find_position(const KeyT& key) {...}
        BucketIterator
        insert_to_position(const ValueT& val, size_t& pos) {...}
```



	X	X	9		88	41	
--	---	---	---	--	----	----	--

Linear Probing : find_position

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



$$h(x) = x \% 13$$

```
class BucketT {  
    pair<KeyT, MappedT> value;  
    unsigned char      status;  
    ...  
}
```

```
vector<BucketT> mBuckets;
```

```
size_t find_position(const KeyT& key) {  
    size_t homePos = hash_to_bucket(key);  
    size_t pos     = homePos;  
    while ( !mBuckets[pos].empty() &&  
            !mEqual(mBuckets[pos].value.first, key) ) {  
        pos = (pos + 1) % mBuckets.size();  
    }  
    return pos;  
}
```

$$\text{ถ้า } \lambda = \frac{mUsed}{bucket_count} < 1$$

ต้องมีช่องว่างแน่ๆ

ต้องมั่นใจว่า จะพบ empty() หรือไม่ก็ พบ key

insert

```
BucketIterator insert_to_position(const ValueT& val, size_t& pos) {  
    if (load_factor() > max_load_factor()) {  
        rehash(2*bucket_count());  
        pos = find_position(val.first);  
    }  
    mSize++;  
    mBuckets[pos].value = val;  
    if (mBuckets[pos].empty()) mUsed++;  
    mBuckets[pos].mark_data();  
    return mBuckets.begin()+pos;  
}
```

YZ, 9

pos



```
pair<iterator, bool> insert(const ValueT& val) {  
    pair<iterator, bool> result;  
    size_t pos = find_position(val.first);  
    if (mBuckets[pos].available()) {  
        BucketIterator it = insert_to_position(val, pos);  
        result.first = iterator(it, mBuckets.end());  
        result.second = true;  
    } else {  
        result.first = iterator(mBuckets.begin()+pos, mBuckets.end());  
        result.second = false;  
    }  
    return result;  
}
```

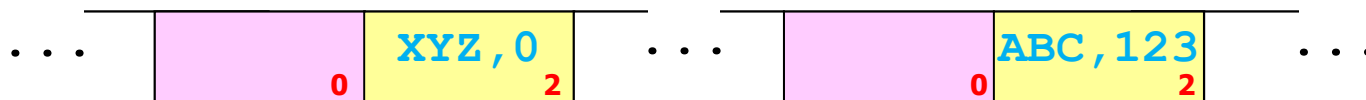
ใช้ #deleted + #data คำนวณ load factor

```
class unordered_map {
    ...
    vector<BucketT> mBuckets;
    size_t           mSize;
    HasherT          mHasher;
    EqualT           mEqual;
    float            mMaxLoadFactor;
    size_t           mUsed;           // # data + # deleted
public:
    float load_factor() {
        return (float)mUsed/mBuckets.size();
    }
    float max_load_factor() {
        return mMaxLoadFactor;
    }
    void max_load_factor(float z) {
        mMaxLoadFactor = z;
        rehash(bucket_count());
    }
}
```

operator []

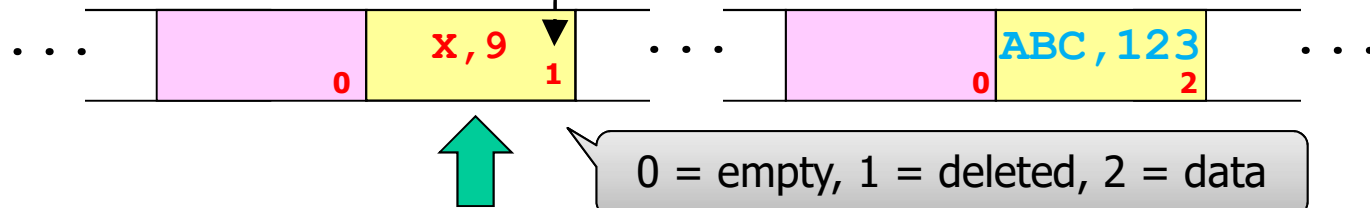
```
MappedT& operator[](const KeyT& key) {  
    size_t pos = find_position(key);  
    if (mBuckets[pos].available()) { // ไม่มีข้อมูล  
        insert_to_position(make_pair(key, MappedT()), pos);  
    }  
    return mBuckets[pos].value.second;  
}
```

```
CP::unordered_map<string,int> m;  
m["ABC"] = 123;  
cout << m.size() << endl; // 1  
cout << m["ABC"] << "," << m["XYZ"] << endl;  
cout << m.size() << endl; // 2
```



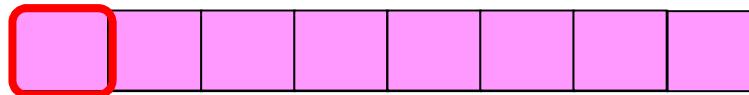
erase

```
size_t erase(const KeyT & key) {  
    size_t pos = find_position(key);  
    if ( mBuckets[pos].has_data() ) {  
        mBuckets[pos].mark_deleted();  
        mSize--;  
        return 1;  
    } else {  
        return 0;  
    }  
}
```



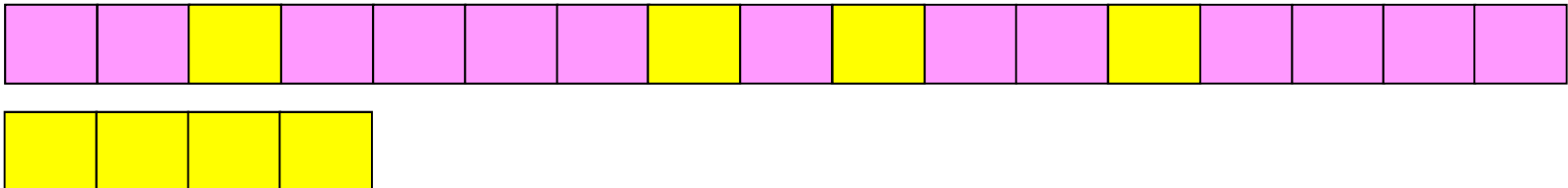
clear

```
void clear() {  
    for (auto& bucket : mBuckets) {  
        bucket.mark_empty();  
    }  
    mSize = 0;  
    mUsed = 0;  
}
```



rehash

```
void rehash(size_t m) {  
    if (load_factor() <= max_load_factor() &&  
        m <= mBuckets.size()) return;  
    m = max(m, (size_t)(0.5+mSize/mMaxLoadFactor));  
    m = *lower_bound(PRIMES, PRIMES+N_PRIMES, m);  
    vector<ValueT> tmp;  
    for (auto& val : *this) {  
        tmp.push_back(val);  
    }  
    this->clear();  
    mBuckets.resize(m);  
    for (auto& val : tmp) {  
        this->insert(val);  
    }  
}
```



บริการอื่นๆ

```
class unordered_map {
    ...
public:
    bool    empty()                { return mSize == 0; }
    size_t  size()                 { return mSize; }
    size_t  bucket_count()         { return mBuckets.size(); }
    size_t  bucket_size(size_t n) {
        return mBuckets[n].has_data() ? 1 : 0
    }
    float   load_factor() {
        return (float)mUsed/mBuckets.size();
    }
    float   max_load_factor() {
        return mMaxLoadFactor;
    }
    void     max_load_factor(float z) {
        mMaxLoadFactor = z;
        rehash(bucket_count());
    }
}
```

การเกาะกลุ่มปฐมภูมิ (Primary Clustering)

- ถ้าใช้ linear probing แล้วเพิ่มข้อมูลตัวใหม่อีกตัว ลงตารางข้างล่างนี้ อยากทราบว่า ข้อมูลใหม่นี้จะ ถูกนำไปเก็บไว้ที่ช่องใดด้วยความน่าจะเป็นสูงสุด



Cookie Monster
Effect

การตรวจกำลังสอง (Quadratic Probing)

- เพื่อจัดการเกาะกลุ่มปฏิกิริยา
- หลีกเลี่ยงการตรวจช่องติด ๆ กัน
- ให้ตรวจแบบก้าวกระโดดห่าง ๆ

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

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

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

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

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

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

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

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

Linear vs. Quadratic

```
size_t find_position(const KeyT& key) {  
    size_t homePos = hash_to_bucket(key);  
    size_t pos = homePos, m = mBuckets.size();  
    while ( !mBuckets[pos].empty() &&  
            !mEqual(mBuckets[pos].value.first, key) ) {  
  
        pos = (pos + 1) % m;  
    }  
    return pos;  
}
```

$$h_j(x) = (h(x) + 1) \% m$$

```
size_t find_position(const KeyT& key) {  
    size_t homePos = hash_to_bucket(key);  
    size_t pos = homePos, m = mBucket.size(), col_count = 0;  
    while ( !mBuckets[pos].empty() &&  
            !mEqual(mBuckets[pos].value.first, key) ) {  
  
        col_count++;  
        pos = (pos + 2*col_count-1) % m;  
    }  
    return pos;  
}
```

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

Linear vs. Quadratic

```
size_t find_position(const KeyT& key) {  
    size_t homePos = hash_to_bucket(key);  
    size_t pos = homePos, m = mBuckets.size(), col_count = 0;  
    while ( !mBuckets[pos].empty() &&  
            !mEqual(mBuckets[pos].value.first, key) ) {  
        col_count++;  
        pos = (homePos + col_count) % m;  
    }  
    return pos;  
}
```

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

```
size_t find_position(const KeyT& key) {  
    size_t homePos = hash_to_bucket(key);  
    size_t pos = homePos, m = mBuckets.size(), col_count = 0;  
    while ( !mBuckets[pos].empty() &&  
            !mEqual(mBuckets[pos].value.first, key) ) {  
        col_count++;  
        pos = (homePos + col_count*col_count) % m;  
    }  
    return pos;  
}
```

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

คลาสเพื่อการคำนวณตำแหน่งถัดไป

```
class LinearProbing {  
public:  
    size_t operator() ( size_t home_pos,  
                        size_t col_count,  
                        size_t bucket_count ) {  
        return (home_pos + col_count) % bucket_count;  
    }  
};
```

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

```
LinearProbing mNextAddress;  
...  
size_t find_position(const KeyT& key) {  
    size_t homePos = hash_to_bucket(key);  
    size_t pos = homePos, m = mBuckets.size(), col_count = 0;  
    while ( !mBuckets[pos].empty() &&  
            !mEqual(mBuckets[pos].value.first, key) ) {  
        col_count++;  
        pos = mNextAddress(homePos, col_count, m);  
    }  
    return pos;  
}
```

คลาสเพื่อการคำนวณตำแหน่งถัดไป

```
class QuadraticProbing {  
public:  
    size_t operator() ( size_t home_pos,  
                        size_t col_count,  
                        size_t bucket_count ) {  
        return (home_pos + col_count*col_count) % bucket_count;  
    }  
};
```

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

```
QuadraticProbing mNextAddress;  
...  
size_t find_position(const KeyT& key) {  
    size_t homePos = hash_to_bucket(key);  
    size_t pos = homePos, m = mBuckets.size(), col_count = 0;  
    while ( !mBuckets[pos].empty() &&  
            !mEqual(mBuckets[pos].value.first, key) ) {  
        col_count++;  
        pos = mNextAddress(homePos, col_count, m);  
    }  
    return pos;  
}
```

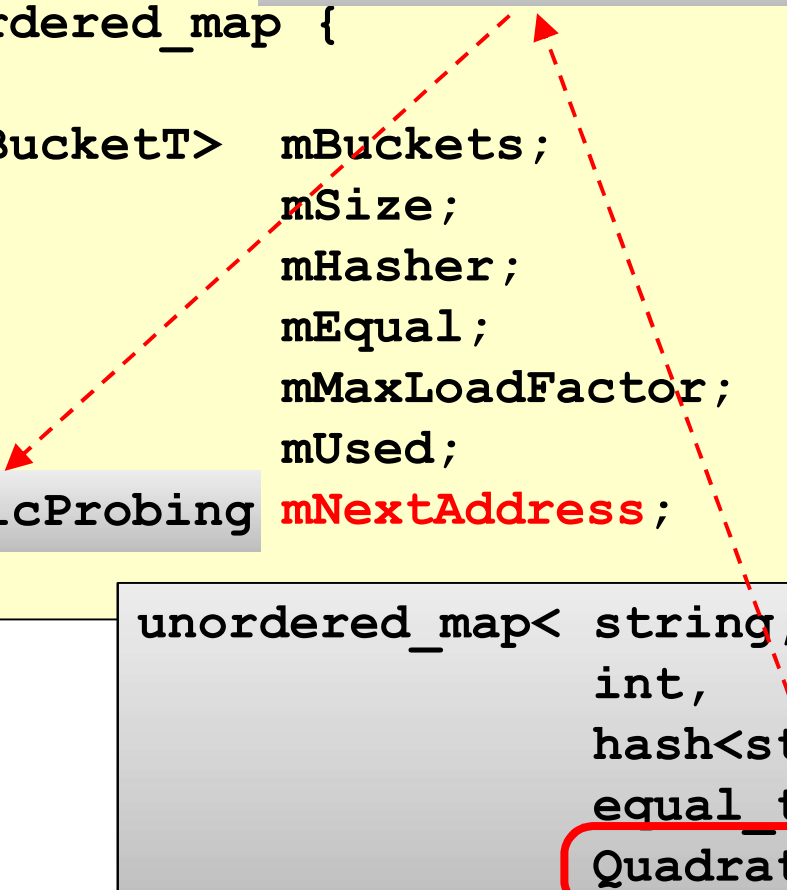
LinearProbing vs. QuadraticProbing

```
class LinearProbing {
public:
    size_t operator() ( size_t home_pos,
                        size_t col_count,
                        size_t bucket_count) {
        return (home_pos + col_count) % bucket_count;
    }
};
```

```
class QuadraticProbing {
public:
    size_t operator() ( size_t home_pos,
                        size_t col_count,
                        size_t bucket_count) {
        return (home_pos + col_count*col_count) % bucket_count;
    }
};
```

NextAddressT

```
template <typename KeyT,  
          typename MappedT,  
          typename HasherT = std::hash<KeyT>,  
          typename EqualT = std::equal_to<KeyT>,  
          typename QuadraticProbing >  
class unordered_map {  
    ...  
    vector<BucketT> mBuckets;  
    size_t mSize;  
    HasherT mHasher;  
    EqualT mEqual;  
    float mMaxLoadFactor;  
    size_t mUsed;  
    QuadraticProbing mNextAddress;  
};
```



```
unordered_map< string,  
              int,  
              hash<string>,  
              equal_to<string>,  
              QuadraticProbing > mymap;
```

default constructor

```
class unordered_map {  
    ...  
    vector<BucketT>    mBuckets;  
    size_t              mSize;  
    HasherT            mHasher;  
    EqualT             mEqual;  
    float              mMaxLoadFactor;  
    size_t              mUsed;  
    NextAddressT        mNextAddress;  
  
    unordered_map( ) :  
  
        mBuckets(vector<BucketT>(11)), mSize(0),  
        mHasher(HasherT()), mEqual(EqualT()),  
        mMaxLoadFactor(0.5), mUsed(0),  
        mNextAddress( NextAddressT() )  
  
    { }  
};
```


copy constructor

```
class unordered_map {
    ...
    vector<BucketT>    mBuckets;
    size_t             mSize;
    HasherT            mHasher;
    EqualT             mEqual;
    float              mMaxLoadFactor;
    size_t             mUsed;
    NextAddressT       mNextAddress;

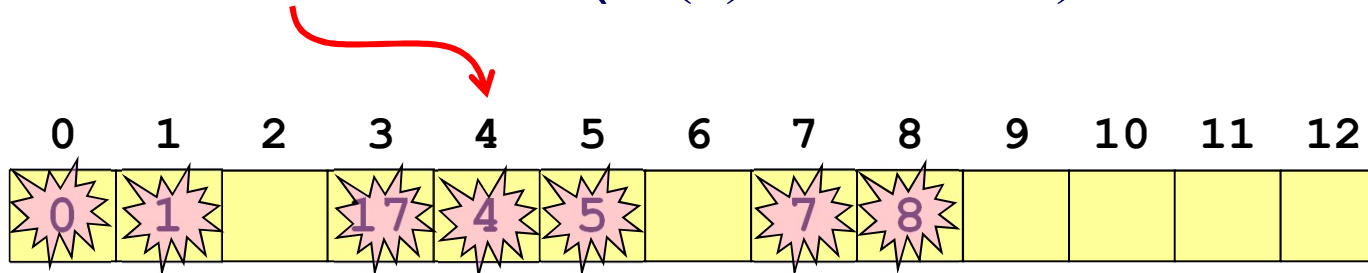
    unordered_map(const
                  unordered_map<KeyT, MappedT, HasherT, EqualT,
                              NextAddressT> &other) :

        mBuckets(other.mBuckets), mSize(other.mSize),
        mHasher(other.mHasher), mEqual(other.mEqual),
        mMaxLoadFactor(other.mMaxLoadFactor), mUsed(other.mUsed),
        mNextAddress( other.mNextAddress )

    { }
```

การตรวจกำลังสองไม่ตรวจทุกช่อง

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



$h(x) = 4$	$(4+7^2) \% 13 = 1$
$(4+1^2) \% 13 = 5$	$(4+8^2) \% 13 = 3$
$(4+2^2) \% 13 = 8$	$(4+9^2) \% 13 = 7$
$(4+3^2) \% 13 = 0$	$(4+10^2) \% 13 = 0$
$(4+4^2) \% 13 = 7$	$(4+11^2) \% 13 = 8$
$(4+5^2) \% 13 = 3$	$(4+12^2) \% 13 = 5$
$(4+6^2) \% 13 = 1$	$(4+13^2) \% 13 = 4$

...

มีช่องว่างอาจหาไม่พบ

เมื่อตารางมีขนาดเป็นจำนวนเฉพาะ

- การตรวจกำลังสองจะดูอย่างน้อยครึ่งหนึ่งของตาราง
- ดังนั้น ถ้า **load factor $\leq 1/2$** ก็สบายใจได้ว่า จะหาช่องว่างพบ เมื่อเพิ่มข้อมูล
- พิสูจน์ : ให้ $0 \leq i < j \leq \lfloor m/2 \rfloor$ ถ้าข้างบนไม่จริง ต้องมีการ probe ครั้งที่ i และ j ที่ดูช่องซ้ำกัน

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

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

mMaxLoadFactor = 0.5

```
class unordered_map {
    ...
    unordered_map( ) :
        mBuckets(vector<BucketT>(11)), mSize(0),
        mHasher(HasherT()), mEqual(EqualT()),
        mMaxLoadFactor(0.5), mUsed(0),
        mNextAddress( NextAddressT() )
    { }
    ...
    size_t find_position(const KeyT& key) {
        size_t homePos = hash_to_bucket(key);
        size_t pos = homePos, m = mBuckets.size(), col_count= 0;
        while ( !mBuckets[pos].empty() &&
                !mEqual(mBuckets[pos].value.first, key) ) {
            col_count++;
            pos = mNextAddress(homePos, col_count, m);
        }
        return pos;
    }
    ...
}
```

ให้ $\lambda_{\max} = 0.5$ ก็มั่นใจได้ว่า
find_position จะพบช่องว่าง

การเกาะกลุ่ม

- การเกาะกลุ่มปฐมภูมิ (primary clustering)
 - เห็นได้ด้วยตา ข้อมูลอยู่ติด ๆ กัน
 - กลุ่มที่โต ย่อมมีโอกาสโตขึ้น
 - 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

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

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

เปรียบเทียบเวลาการทำงาน (จาวา)

$1117 = 1 \times 3 \times 3 \times 3 / 2 \times 3 \times 3 \times 3 \times 3 \times 3 / 2 / 2 \times 3 \times 3 / 2 / 2 / 2 / 2 \times 3 \times 3 \times 3 / 2 / 2 / 2 \times 3 / 2$

```
public static void main(String[] args) {
```

```
    Set set = new ArraySet();
```

```
    ArraySet,
```

```
    Queue q = new ArrayQueue();
```

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

```
}
```

ตอนทำงานเสร็จ set มีข้อมูลจำนวน 73816 ตัว

ข้อควรระวัง

- ไม่เหมาะกับบริการที่
 - แจกแจงข้อมูลด้วย iterator
 - เกี่ยวข้องกับอันดับของข้อมูล getMin, getMax, ...
 - ต้องค้นทั้งตาราง $\Theta(m+n)$
- ต้องระวังเรื่องฟังก์ชันแฮช
 - ฟังก์ชันแฮชไม่ดี ก็ใช้งานได้ แต่ถ้า n มาก อาจช้าเป็น $O(n)$

```
class BookHasher {  
public:  
    size_t operator() (const Book& b) const {  
        return 0;  
    }  
};
```

สรุป

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