Hash Table

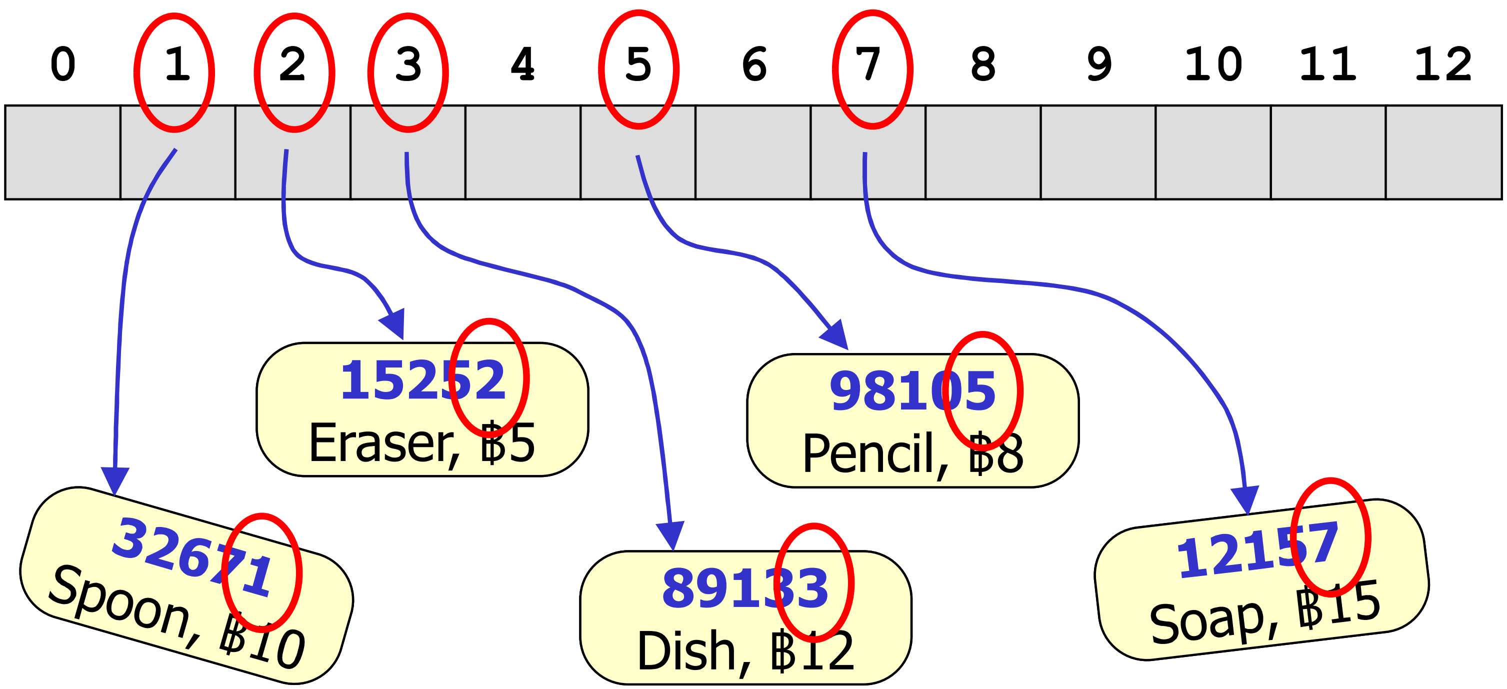
สมชาย ประสิทธิ์จูตระกูล
Translated to English by Nuttapong Chentanez

Topics

- Use table to store data with hash function
- Separate chaining
- Hash function
- Hash function considerations
- Hashing in C++
- Open addressing
- Data clustering

Use hash function to compute index

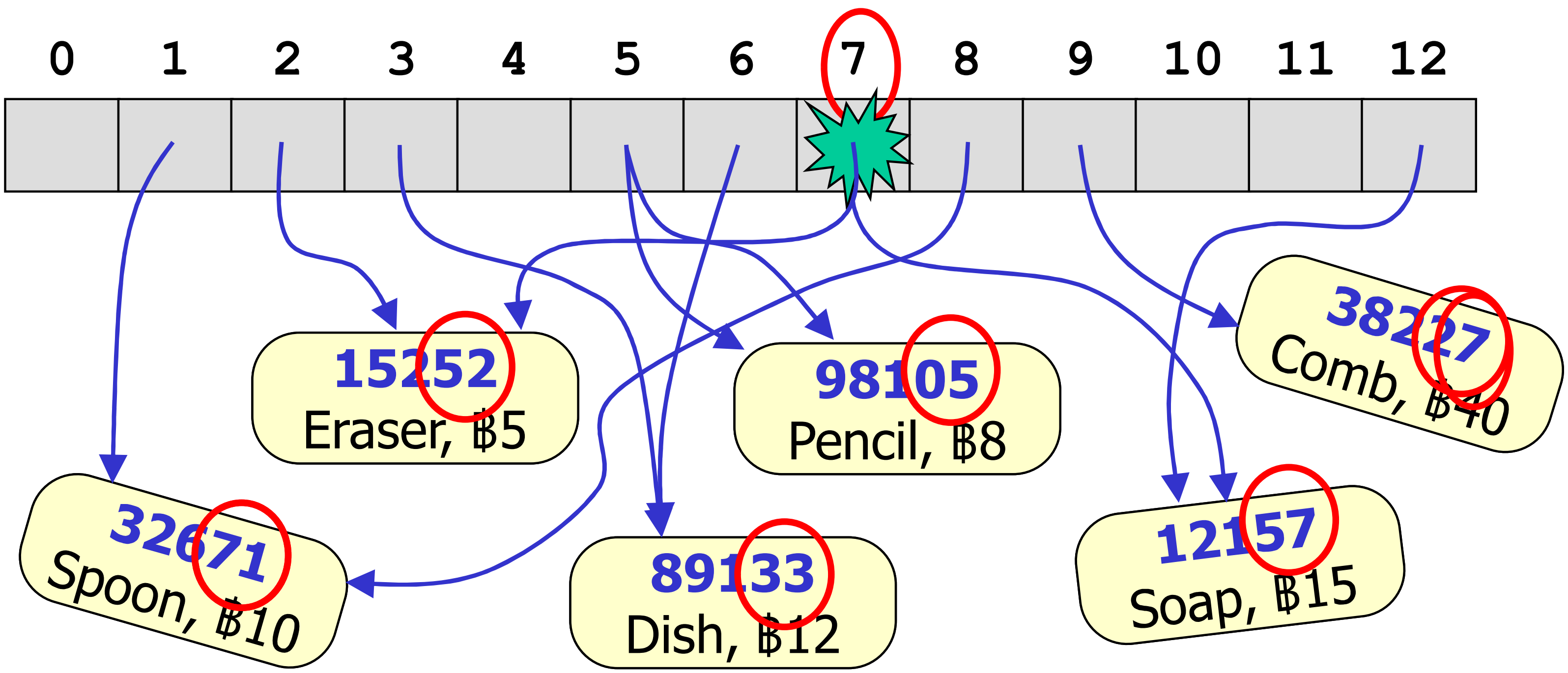
- key of data is what's used for search
- Use table to store data in each slot
- Find f(key) to transform key into index of table
- Can find hash function easily if table is large



`f(key) = key % 10`

Hash function is difficult to find

- When need to store data compactly
- When need to guarantee there's no collision
- If data set is known in advance, could find it
- But in practice, don't know in advance



`f(key) = key % 10`

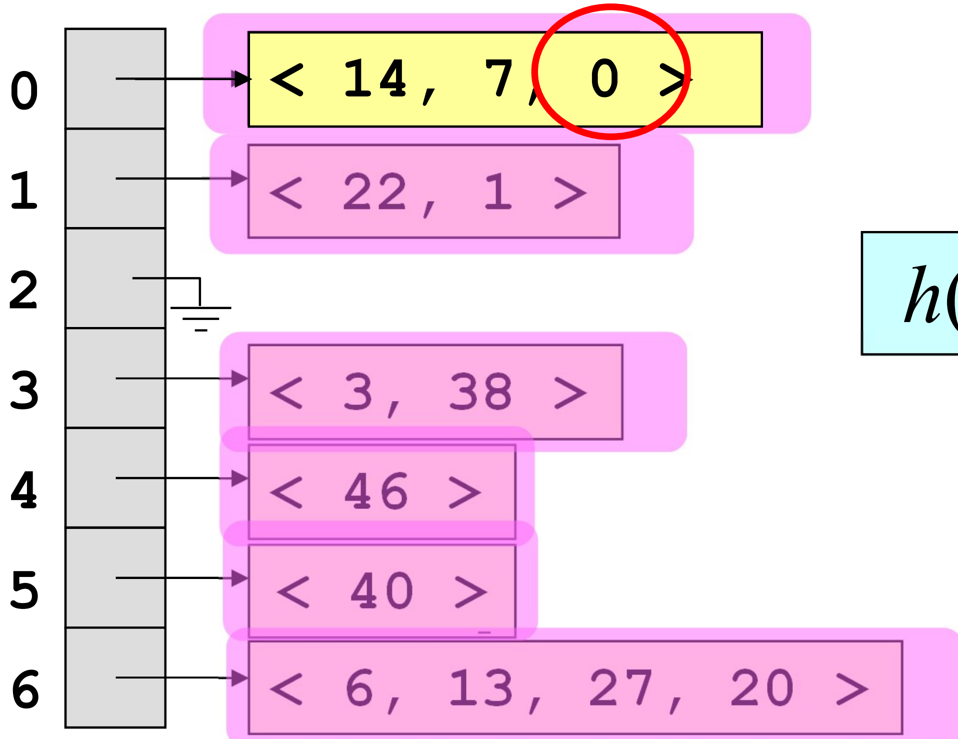
`f(key) = (key/10) % 10`

Change strategy : allow for collision

- So can store data in reasonable size table
- Find way to resolve collision, efficiently

Separate Chaining

- Store the data that collide with each other in the same list



$$h(x) = x \% 7$$

Distribution of data

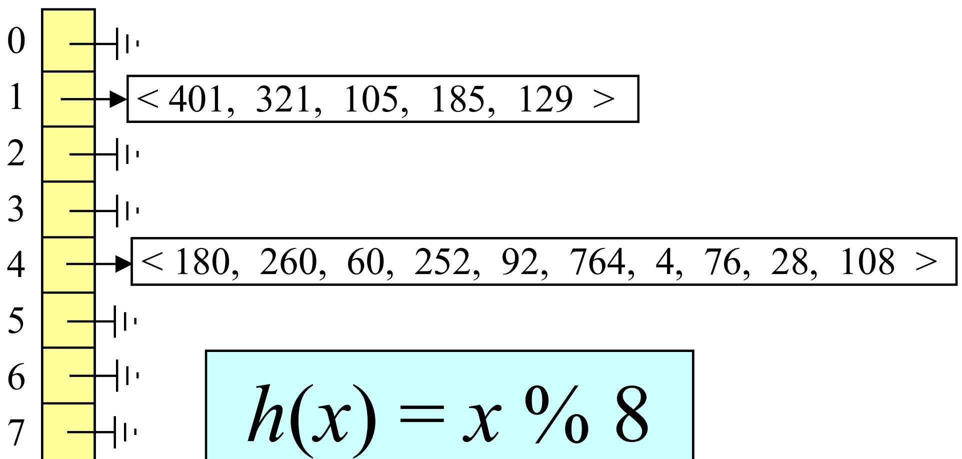
- If data distributes across the table,
  - each list will be of length ≈ λ
  - if λ is small, can search quickly
- If not,
  - some list will be significantly longer than λ
  - slow just like storing in a list

load factor

$$\lambda = n \,/\, m$$

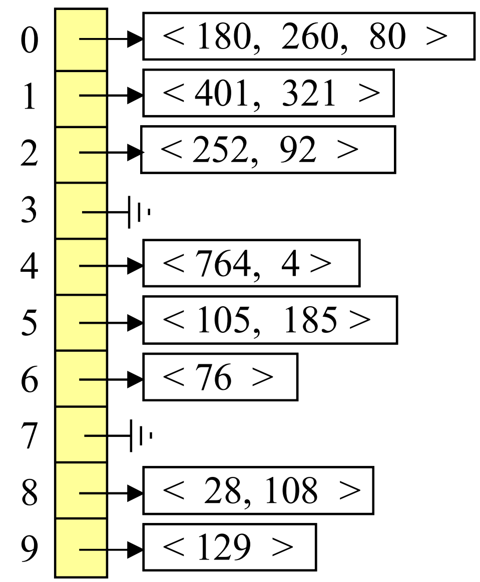
#data

Table size



$$h(x) = x \,\%\, 8$$

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

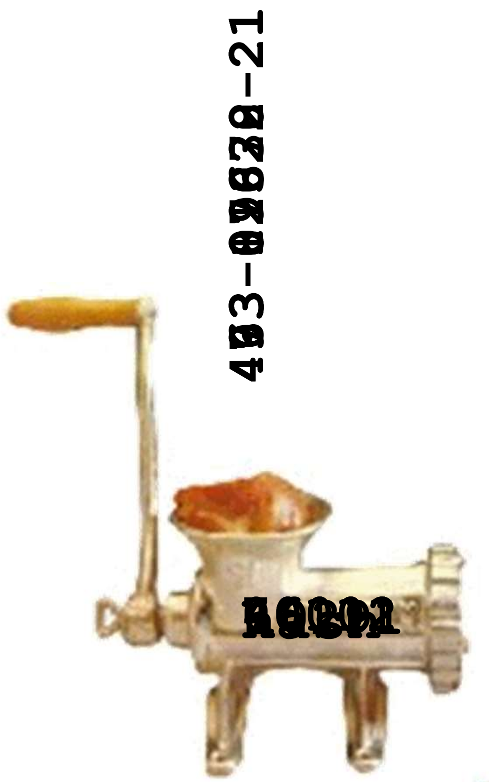


Distribution of data

- Depends on
  - x : key of data
  - h(x) : function to transform key to index
- If the key x is already well distributed
  - If table has 100 slots, let h(x) = x % 100
  - If table has 2^k slots, let h(x) = k right bits of x
- If the key x has some kind of order
  - Student ID, Citizen ID, ...
  - Need to design h(x) to turn x from being in order to being chaotic
  - Call h(x) "Hash function"

Hash Function

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



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

Example of hash functions

```
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

How to make a good hash function?

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

การวิเคราะห์เลขโดด (Digit Analysis)

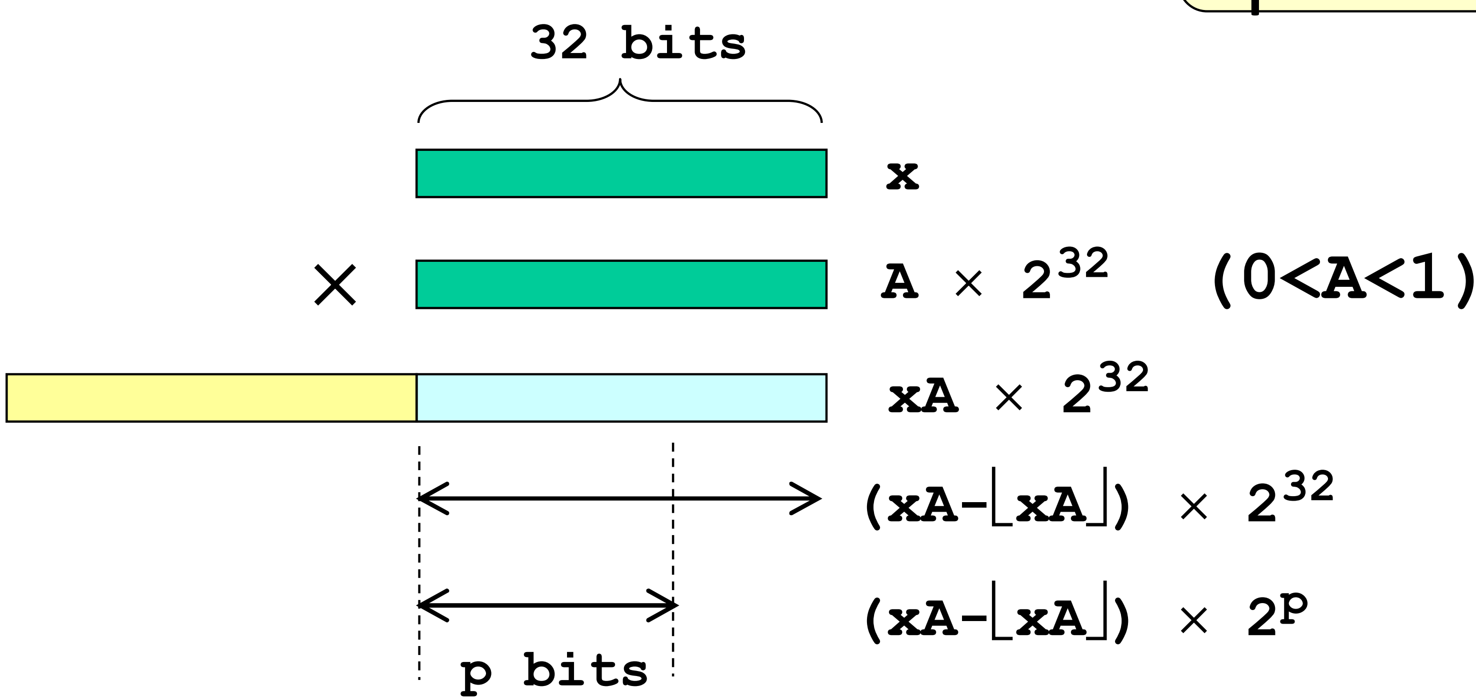
- Take only some digits of the key into consideration
- Ignore those that cause data to not be well distributed
- Example
  - CU Eng student has ID : xx3xxxxx21
  - Remove 3 and 21 from consideration
  - k = 4830109521,
  - k1 = $\lfloor k / 100 \rfloor$ // k1 = 48301095
  - k2 = $\lfloor k1 / 10^6 \rfloor$ // k2 = 48
  - k3 = $k2*10^5 + k1 \% 10^5$ // k3 = 4801095

การคูณ (Multiplicative Hashing)

- Multiply with a real number A between (0,1)
- Fractional part multiply with the size ($m = 2^p$)

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

Fractional part of xA



Fibonacci Hashing

- If A = golden ratio 0.6180339887...
  Can separate nearby keys well

$$\hat{\phi} = \frac{\sqrt{5}-1}{2}$$

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

$0.6180339887 \times 2^{32}$

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

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

การพับ (Folding)

- Separate key into parts and combine (fold) them
- “fold" ≡ +, xor, ...

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

การหาร (Modulus Hashing)

- $h(x) = x \% p$
- Must not choose
  - $p = 10^q$, only use q rightmost digits if key is decimal
  - $p = 2^q$, only choose q rightmost bits
  - p small, not prime number
    - If c is a common divisor of p and x
    - $x \% p$ will be multiple of c
    - If c is small, there's a lot of keys such that $x \% p$ is the multiple of c, which does not distribute well
- In practice, choose p to be prime number!

```
4930102021
4938762821
4731233221
4630987221
```

```
44995961 = 10101011101001010101111001
19436921 = 01001010001001010101111001
24473977 = 01011101010111000101111001
44738937 = 10101010101010100101111001
```

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;
}
```

Any data can be changed into integer

- float $\rightarrow$ integer

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

- String $\rightarrow$ integer
  - Take individual characters and “sum" them
    "DATA" $\rightarrow$ $3x26^3 + 0x26^2 + 19x26^1 + 0x26^0 = 53222$
- class $\rightarrow$ integer
  - Convert each member to integer and then “sum" them

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;
```

Want to use Book as key

```
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;
  }
};
```

Must have operator==

Use Book as key with 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);
    }
  };
}
```

"sum" hash of title and hash of 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;
```

Use Book as key with 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;
}
```

Use Book as key with 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;
```

“Sum"

```
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;
}
```

Want to use Book as key

```
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)
  {}
  ...
};
```

Write 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;
```

การแฮชเอกภพ (Universal Hashing)

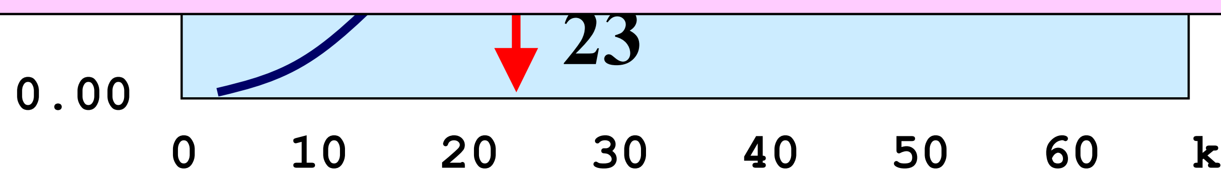
- hash functions we see so far are predictable
  - If data collide a lot now, will forever collide a lot
- Use $h(x) = ((ax + b) \% p) \% m)$
  - $x \in \{0, 1, ..., u-1\}$, u is the number of possible keys
  - m table size
  - Find p , a prime number within $[u, 2u)$
  - $0 < a < p$ and $0 \le b < p$
- Randomly choose a and b when m changes
  - Data that collide a lot now, may collide less in the future
  - Can prove that the average collision is λ

Birthday Paradox

- How many people should there be in a room, so that there's more than 50% change that more than one person has the same birthday
- Person probability of no overlap = $\left(\frac{366}{366}\right)\left(\frac{365}{366}\right)\left(\frac{364}{366}\right)\cdots\left(\frac{366-k+1}{366}\right)$

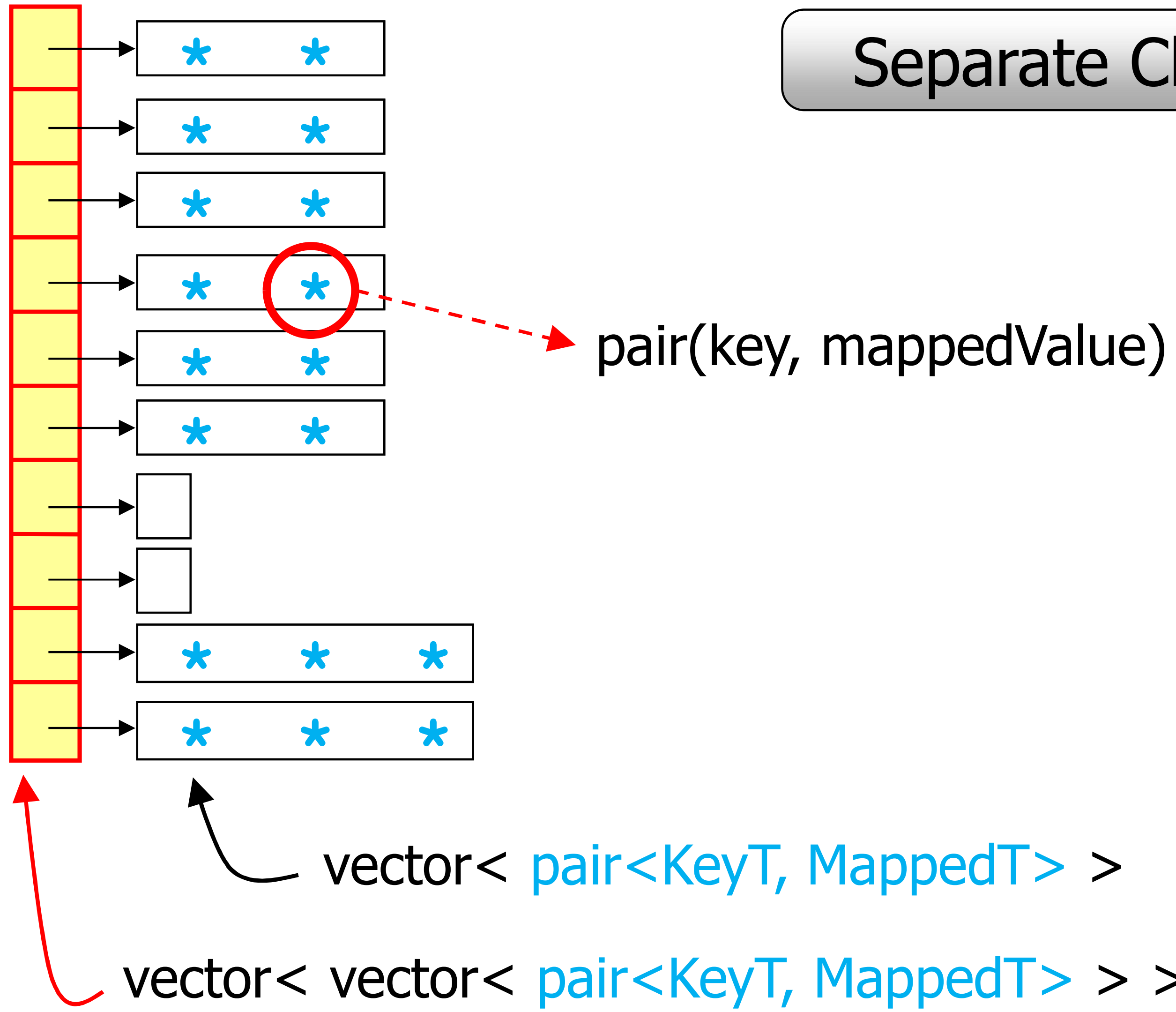
$$1-\left(\left(\frac{366}{366}\right)\left(\frac{365}{366}\right)\left(\frac{364}{366}\right)\cdots\left(\frac{366-k+1}{366}\right)\right)>0.5$$

Person == Data, Birthday == Index in hash table, when hash table has size 366, 23 data is enough for the collision to happen with >50% chance



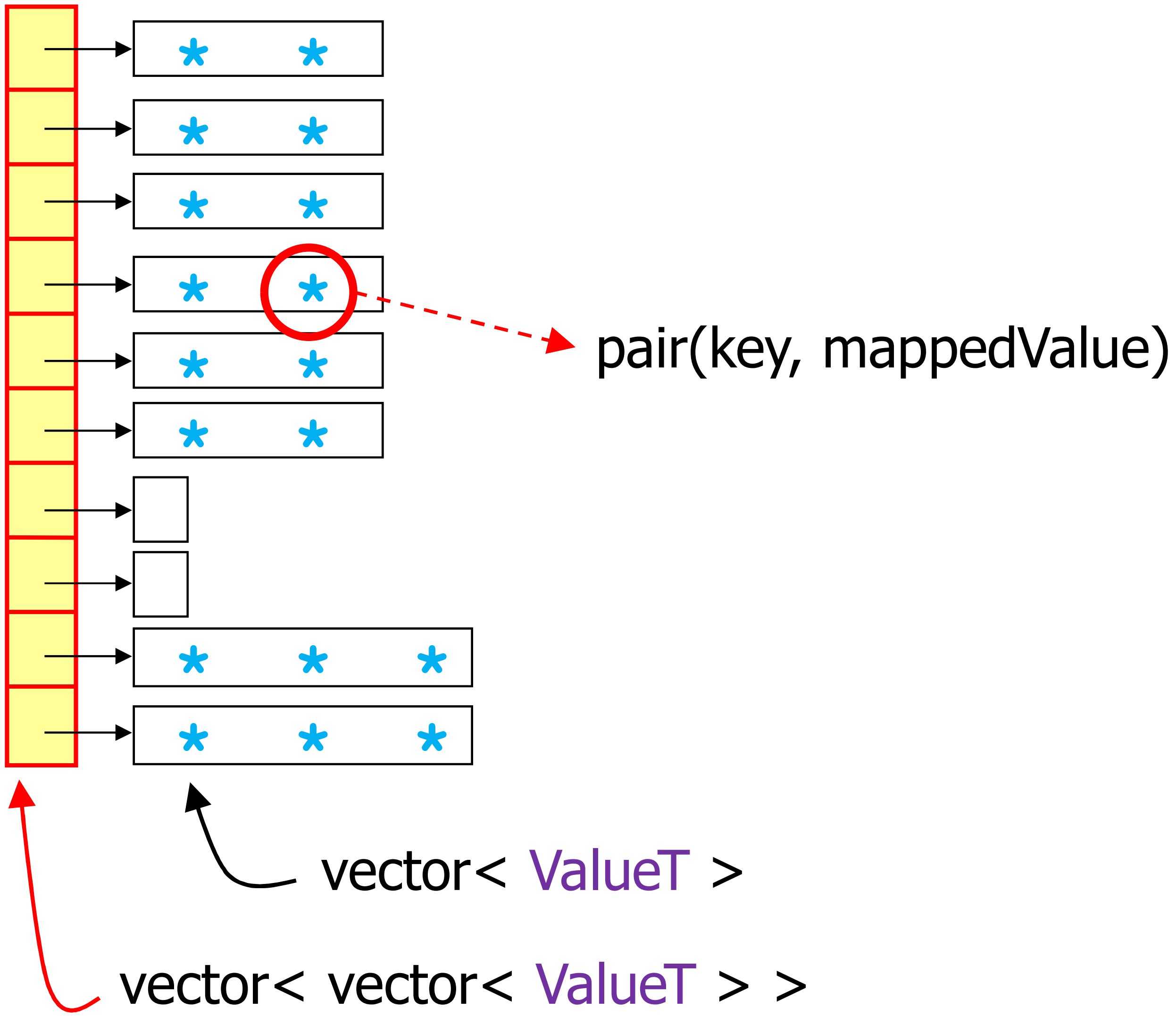
CP::unordered_map<KeyT, MappedT>

Separate Chaining



typedef pair<KeyT,MappedT> ValueT ;

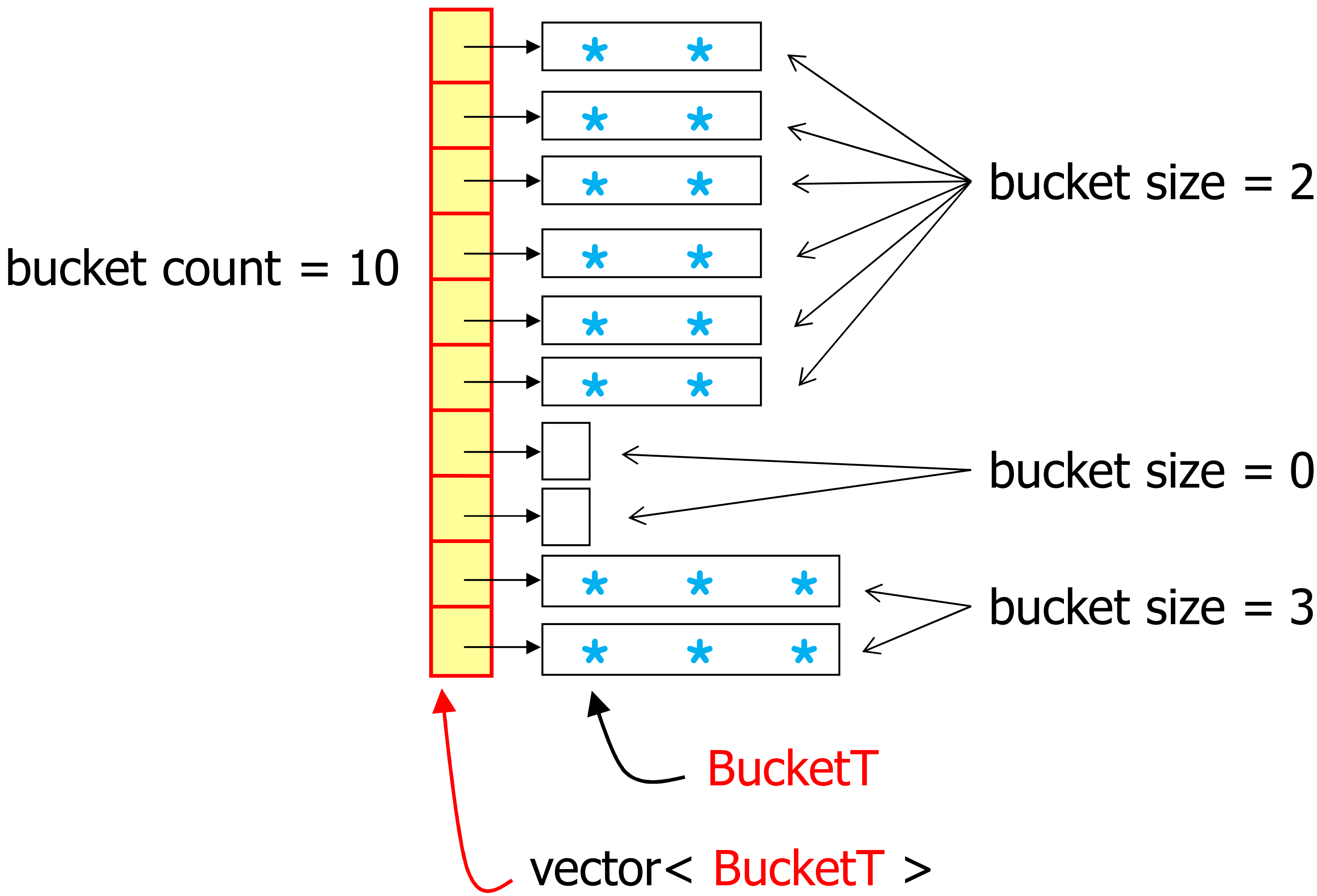
CP::unordered_map<KeyT, MappedT>



typedef pair<KeyT,MappedT> ValueT ;

typedef vector< ValueT > BucketT ;

CP::unordered_map<KeyT, MappedT>

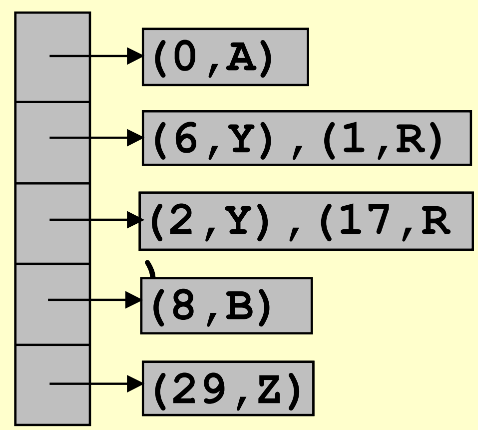


typedef pair<KeyT,MappedT> ValueT ;

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;
```



Use for comparison during key search

hash function for computing index of 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

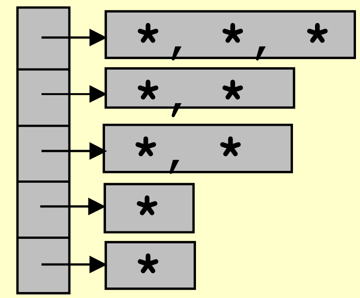
```
template <typename KeyT,
          typename MappedT,
          typename HasherT = std::hash<KeyT>,
          typename EqualT  = std::equal_to<KeyT> >
class unordered_map {
  ...
  std::vector<BucketT> mBuckets;
  size_t               mSize;
  HasherT              mHasher;
  EqualT               mEqual;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

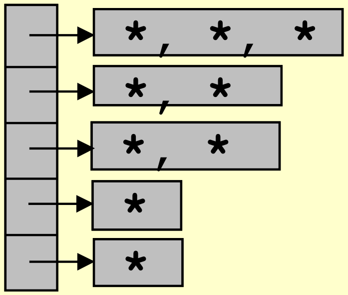
```
template < ... >
class unordered_map {
public:
  bool      empty()                   {...}
  size_t    size()                    {...}
  size_t    bucket_count()            {...}
  size_t    bucket_size(size_t n)     {...}
  float     load_factor()             {...}
  float     max_load_factor()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



$\lambda = 9/5 = 1.8$

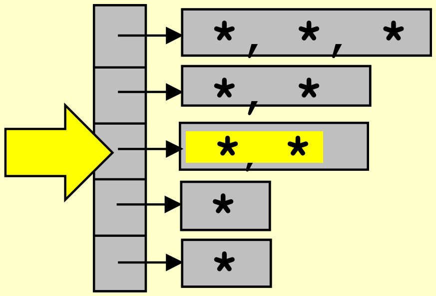
```
m["ok"] = 27;
cout << m["ok"];
```

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);
  // If not found, add pair(key, default value of mapped value)
  return it->second;
}
```

operator[ ], add new entry when not found

```
ValueIterator
insert_to_bucket(const ValueT& val, size_t& bi) {
  if ( table is too congested ) { rehash }
  ++mSize;
  return mBuckets[bi].insert(mBuckets[bi].end(), val);
}

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;
}
```

Result of insert in vector is iterator to the newly added data

Add val to the back

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
  }
}
```

Result of **`erase`** is the number of data erased

- 0 when not found key, no erase took place
- 1 when found key, the key and the mapped value got removed

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 may change if table size changes

iterator of unordered_map

```
ValueIterator insert_to_bucket(const ValueT& val,size_t& bi){
  if ( table is too congested ) { rehash }
  ++mSize;
  return mBuckets[bi].insert(mBuckets[bi].end(), val);
}
```

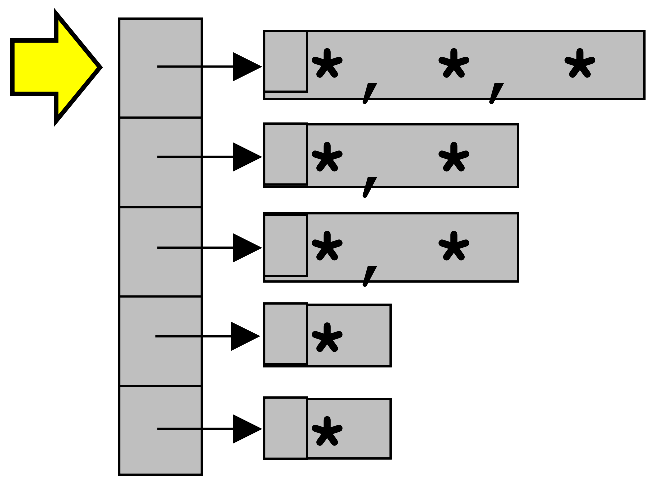
unordered_map: clear

```
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**

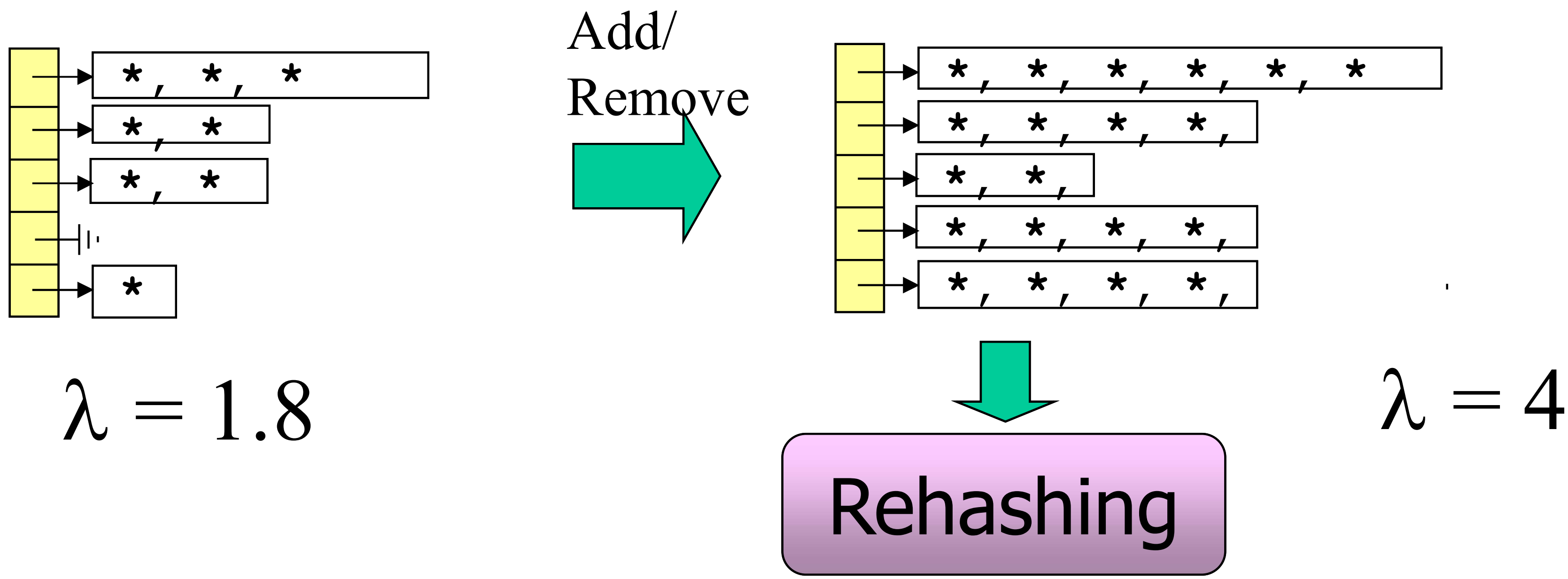
vector<BucketT> mBuckets

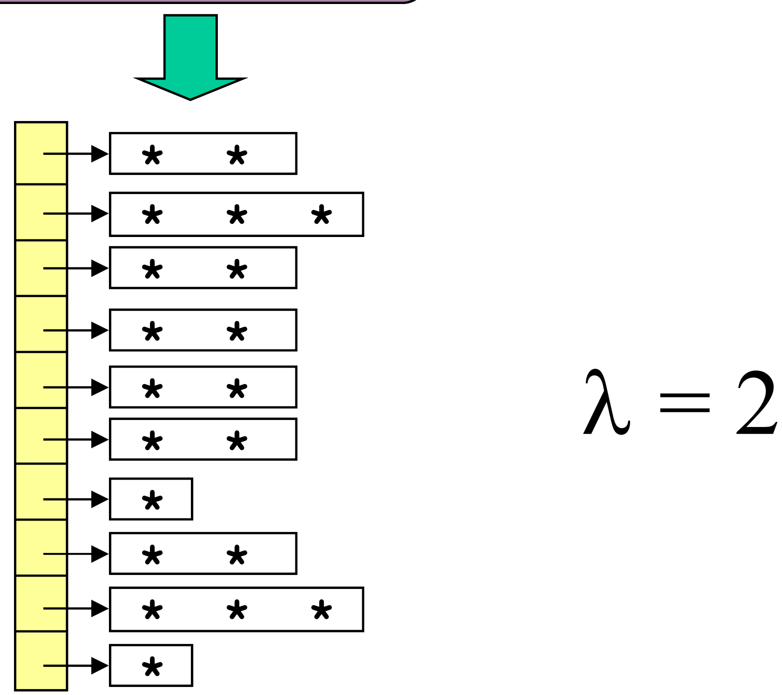


unordered_map: destructor

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

Rehashing





If hash value distributes well,

Remove and search takes O(λ)

If controls λ to not exceed a constant k, add an remove takes constant time!

If “congested” must rehash

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

```
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);
}

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);
}
```

Has to change bi

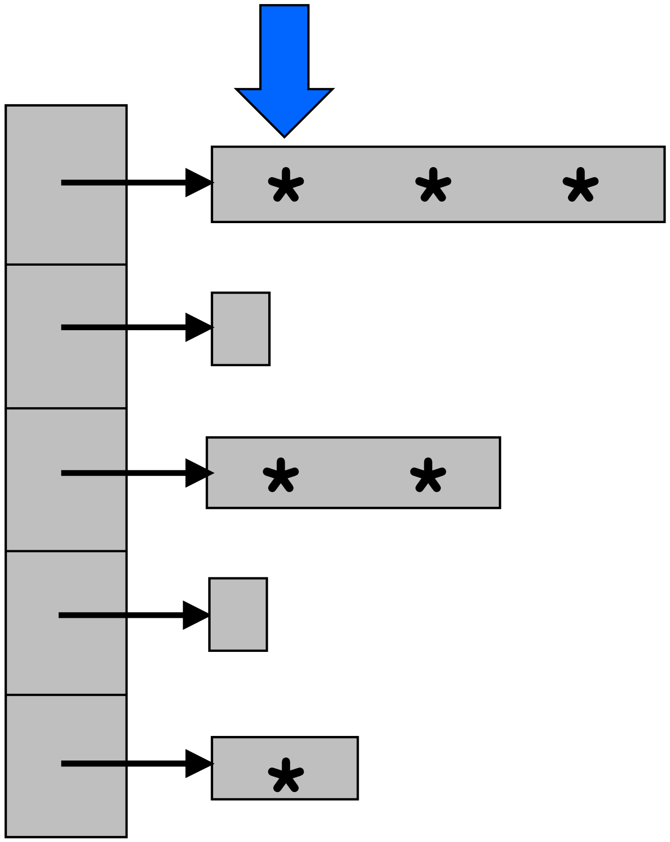
Use prime numbers for table size

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

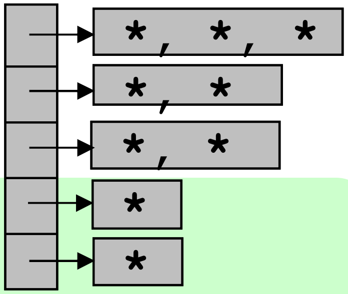
Return the first position in [**PRIMES**, **PRIMES+N_PRIMES**) no smaller than **m**

```
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_map<KeyT,MappedT>::iterator



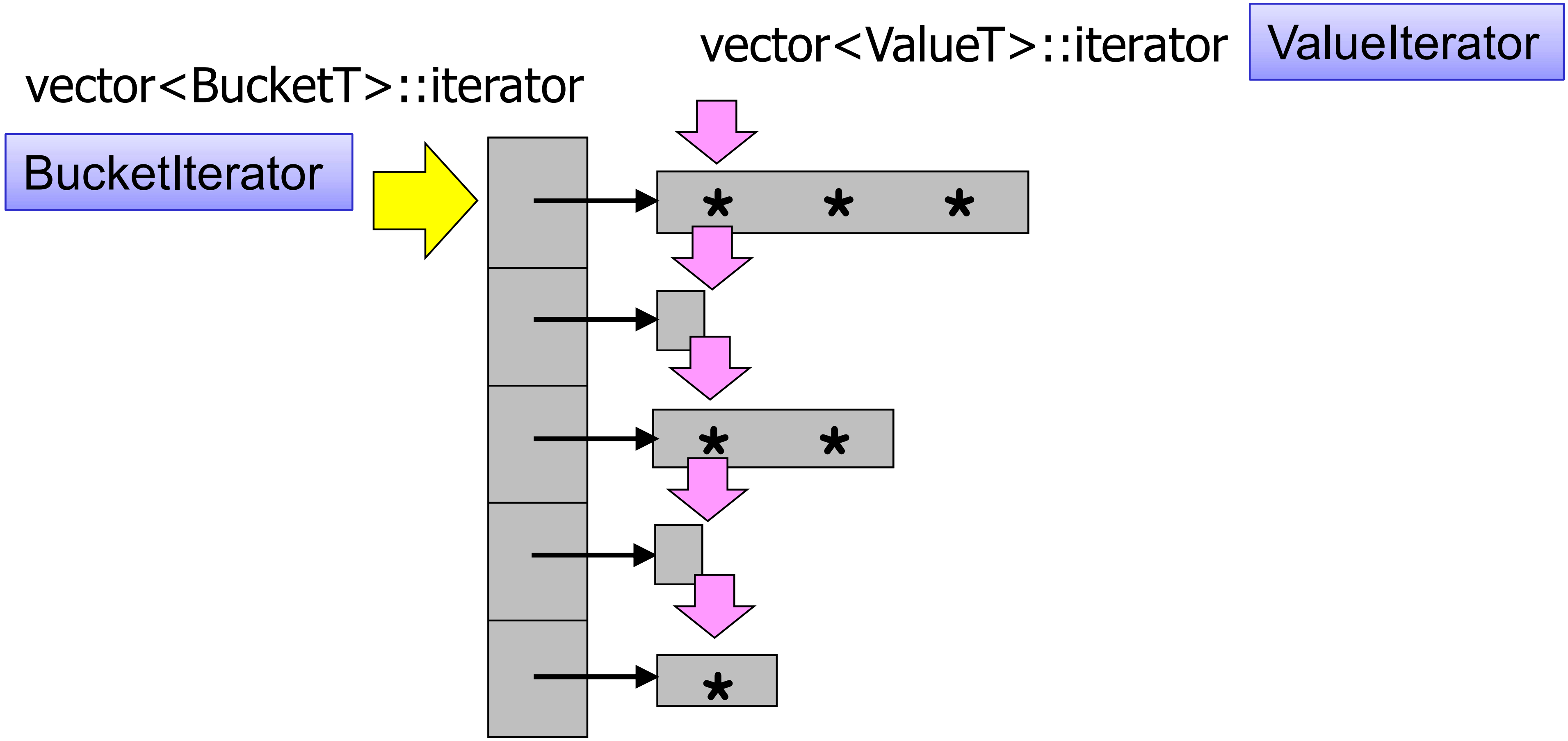
unordered_map<KeyT,MappedT>::iterator



```
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;
  ...
}
```

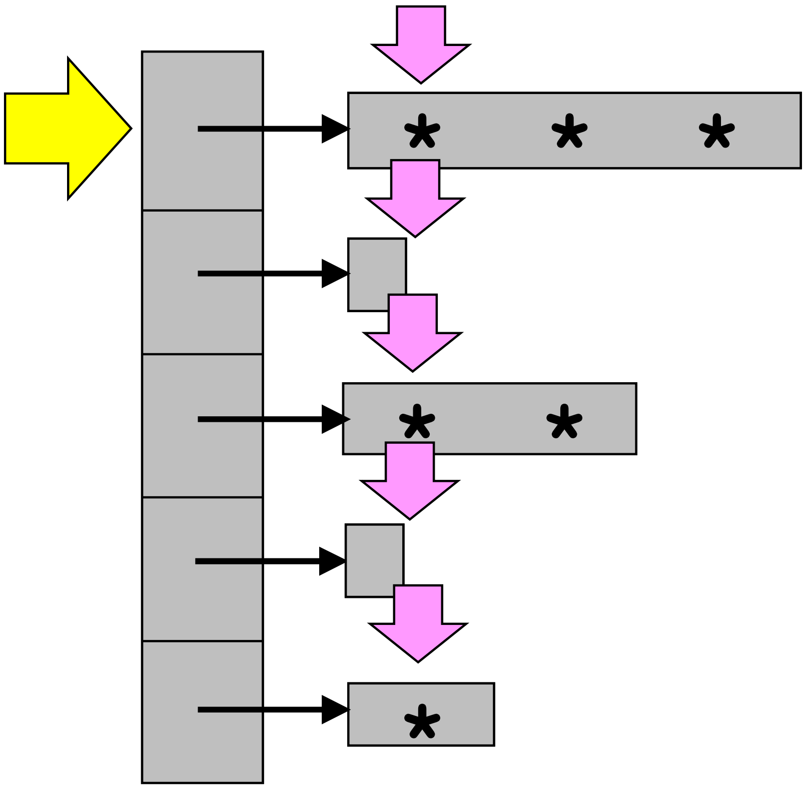
unordered_map<string,int>::iterator it = m.begin();

++iterator




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

++iterator



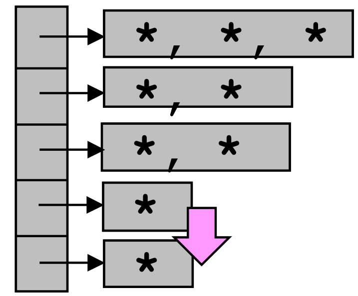
```
class hashtable_iterator {
protected:
  ValueIterator    mCurValueItr;
  BucketIterator  mCurBucketItr;

  void to_next_data( ) {
    while (mCurBucketItr != mBuckets.end() &&
               mCurValueItr  == mCurBucketItr->end()) {
      mCurBucketItr++;
      if (mCurBucketItr == mBuckets.end()) break;
      mCurValueItr = mCurBucketItr->begin();
    }
  }
public:
  hashtable_iterator& operator++( ) {
    mCurValueItr++;
    to_next_data();
    return (*this);
  }
```

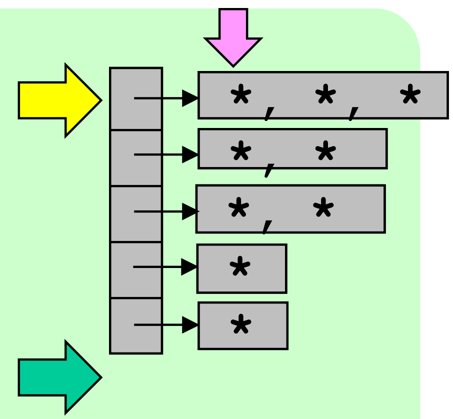
```
it = m.begin();
++it;
...
```

inner class cannot user outer's fields

```
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();
       }
     }
     ...
```

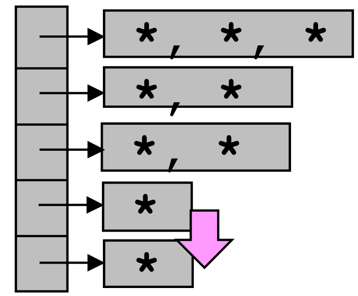
Error!



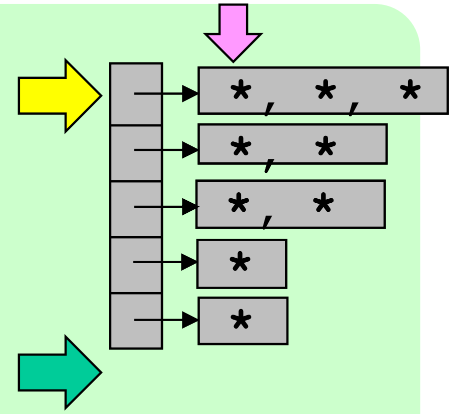


Let **mEndBucketItr** stores **mBuckets.end()** when the iterator is created

iterator has to store mBuckets.end()



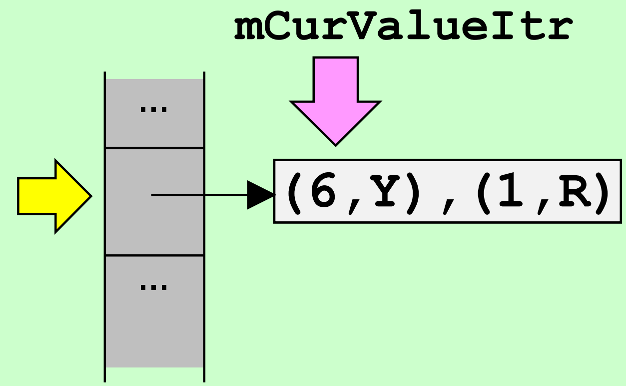

```
class unordered_map {
protected
  vector<BucketT> mBuckets;
  size_t          mSize;
  ...
  class hashtable_iterator {
    ValueIterator  mCurValueItr;
    BucketIterator mCurBucketItr;
    BucketIterator mEndBucketItr;
    ...
    void to_next_data( ) {
      while ( mCurBucketItr != mEndBucketItr &&
              mCurValueItr  == mCurBucketItr->end() ) {
        mCurBucketItr++;
        if (mCurBucketItr == mEndBucketItr) break;
        mCurValueItr = mCurBucketItr->begin();
      }
    }
    ...
```



Let **mEndBucketItr** stores **mBuckets.end()** when the iterator is created

Can either use ++it, it++ ...but

```
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;
    }
    ...
};
```

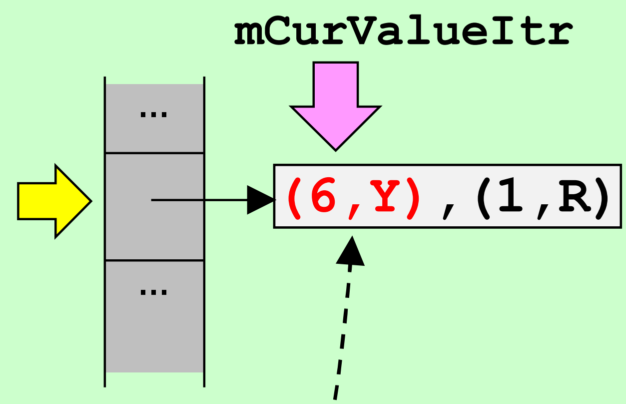


++(++it)   it moves twice

(it++)++   it moves once!

*it and 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);
    }
    ...
};
```



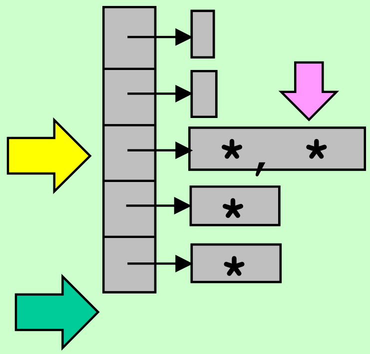
*mCurValueItr is pair (6,Y)

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

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

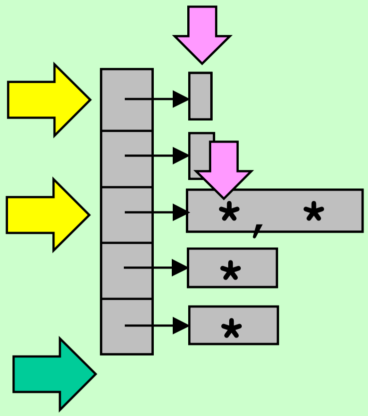
it1 == it2 and it1 != it2

```
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;
    }
    ...
};
```



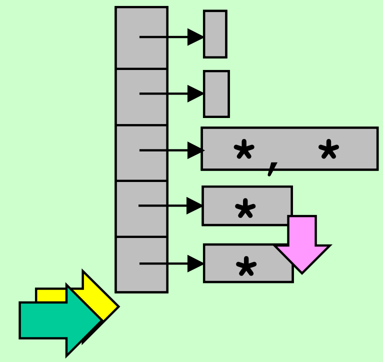
hashtable_iterator :: constructor

```
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()



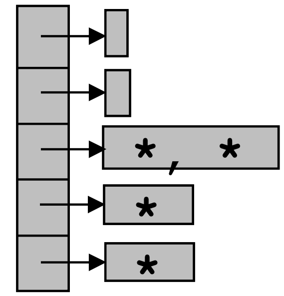

```
class hashtable_iterator {
 protected:
   ValueIterator    mCurValueItr;
   BucketIterator   mCurBucketItr;
   BucketIterator   mEndBucketItr;
 public:
   hashtable_iterator(ValueIterator  valueItr,
                      BucketIterator bucketItr,
                      BucketIterator endBucketItr)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

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

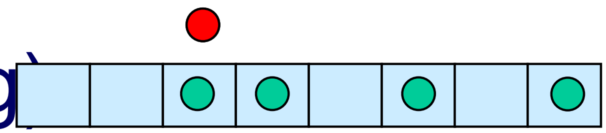
Don’t forget default constructor

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

Other ways to resolve collision



- แบบแยกกันโยง (separate chaining)
  - Each table's entry is a vector of data
  - Data with same hash value stored together, not affect others
- แบบเลขที่อยู่เปิด (open addressing)



  - Each entry store data
  - If collide, find a new free entry in the table to store the data
  - $\lambda = n/m \leq 1$ all the time, in practice ($\lambda \leq 0.5$)
  - Many ways to find the new entry when there's collion
    - การตรวจเชิงเส้น (linear probing)
    - การตรวจกำลังสอง (quadratic probing)
    - การตรวจสองชั้น (double hashing)

การตรวจเชิงเส้น (Linear Probing)

- When collide find the empty slot by keep looking at the next entries
- Let $h_j(x)$ be the index to probe after colliding j times
- $h_0(x) = h(x)$ is the first entry to look (home address)

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

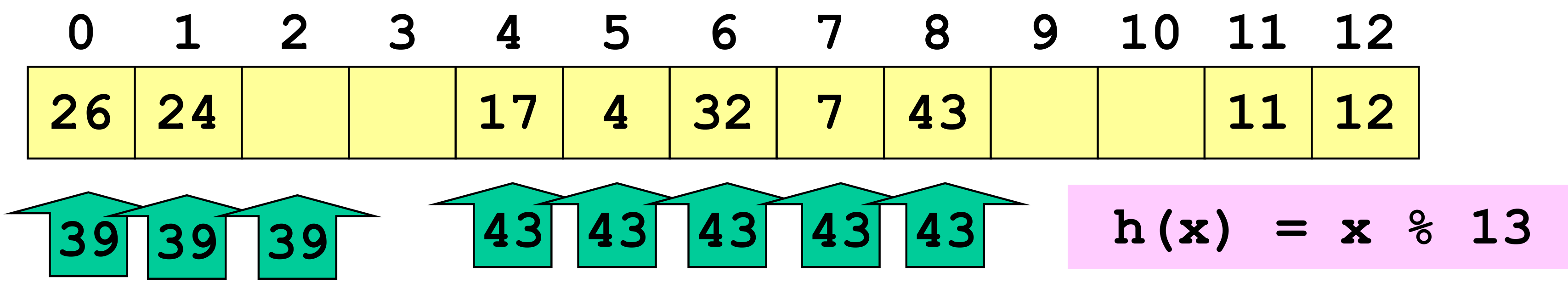
$$h_j(x) = (h_{j-1}(x) + 1)\ \%\ m$$

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

Use **h(x) = x % 13** add data with the following keys

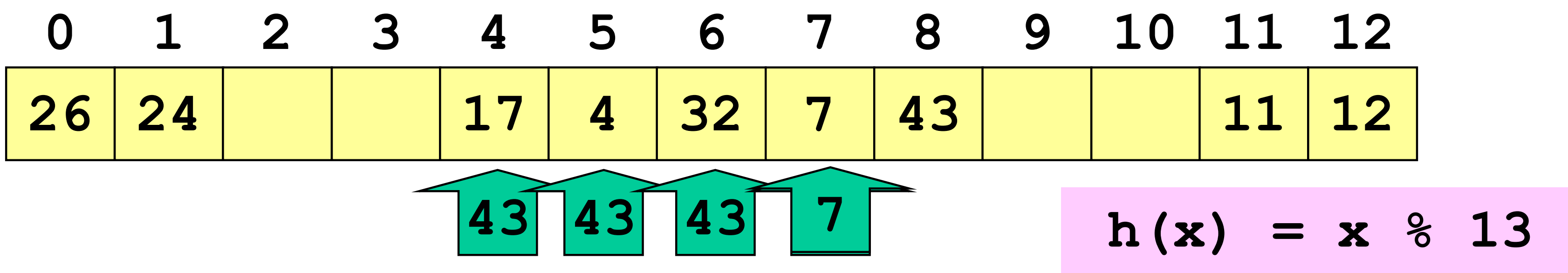
17 32 26 7 4 43 12 11 24

Linear Probing : Search



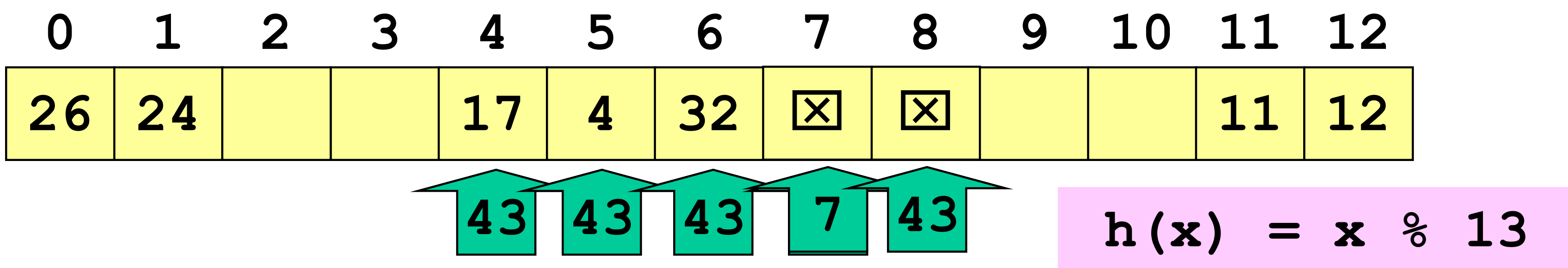
Not found when an empty slot is encountered

Linear Probing : erase



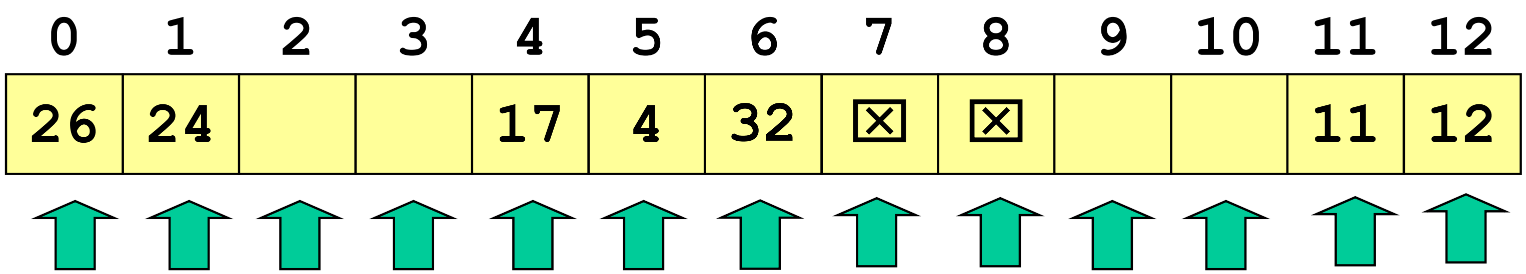
Won’t find 43 because stop looking, even though 43 exists

Linear Probing : erase



Status of each slot

- Each slot has 3 states
  - 0 : empty : Empty never store data
  - 1 : deleted : Store deleted data
  - 2 : data : Store data



Data stored in the table

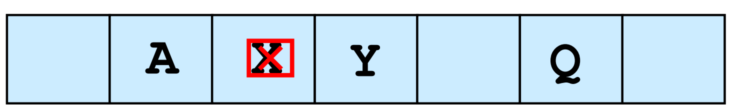
	0	1	2	3	4	5	6	7	8	9	10	11	12
mBuckets	26,? 2	24,? 2	0	0	17,? 2	4,? 2	?,? 1	?,? 1	43,? 2	0	0	11,? 2	12,? 2

```
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

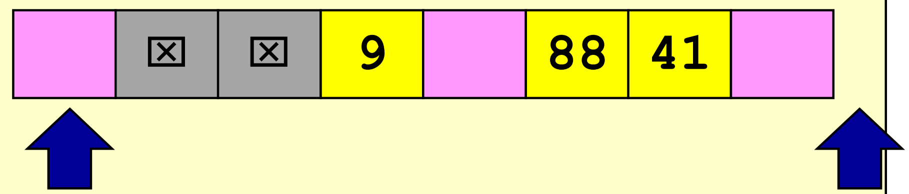
Changing the status of the bucket

- 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



Look at iterator

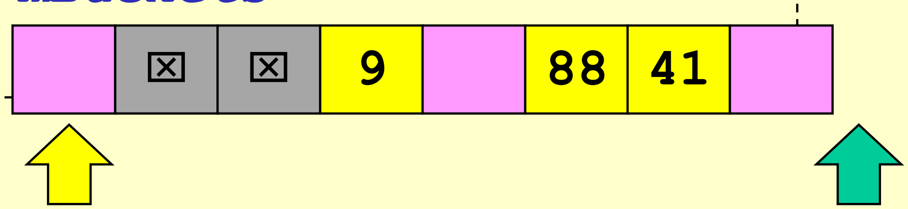
mBuckets




```
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()   {...}
   ...
```

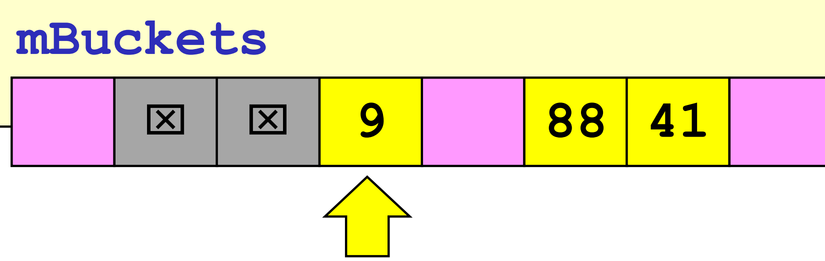
```
class unordered_map {
  ...
 typedef typename vector<BucketT>::iterator BucketIterator;
 class hashtable_iterator {
    protected:
      BucketIterator mCurBucketItr;
      BucketIterator mEndBucketItr;
      void to_next_data() {
        while ( mCurBucketItr != mEndBucketItr &&
                !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 and 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;
    }
```



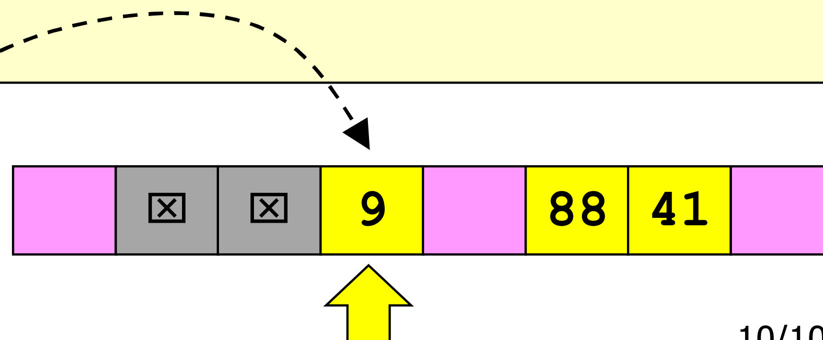
*it and it->

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

  public:
    ...
    ValueT & operator*() {
      return mCurBucketItr->value;
    }

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

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

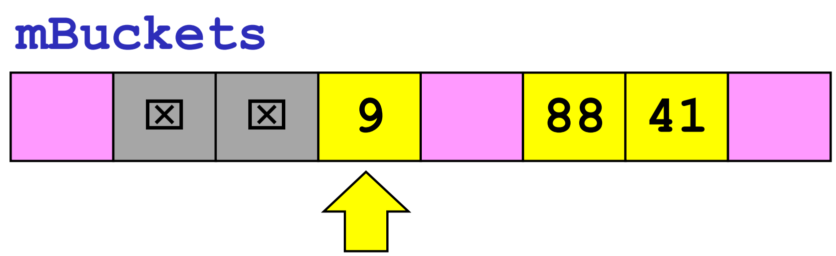
```
it = m.begin();
it->second = 78;
```

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

};
```

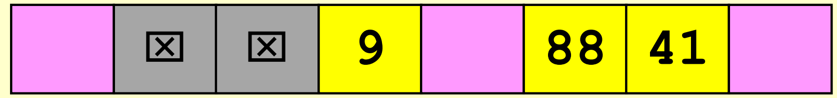


== and !=

```
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);
    }
```



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;         // Use in hash_to_bucket
    EqualT          mEqual;          // Use in find_position
    float           mMaxLoadFactor; // Use in 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) {...}
```

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

43

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;
}
```

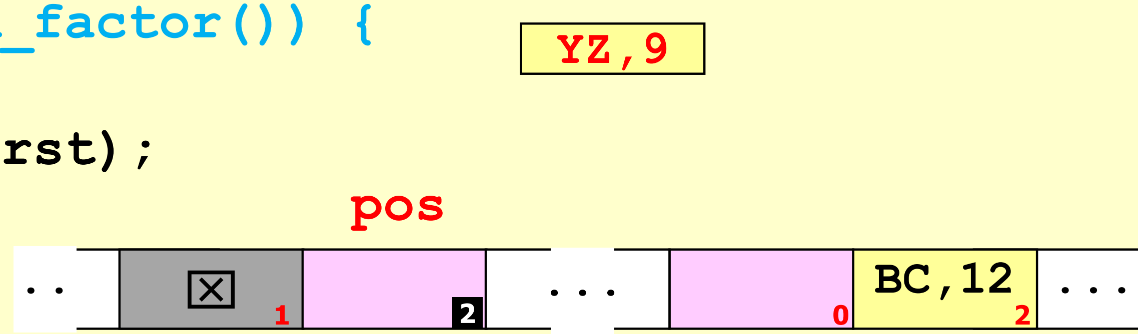
If $\lambda = \dfrac{mUsed}{bucket\_count} < 1$

,must have an empty slot

Must be sure to find empty() or 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;
}
```




```
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;
}
```

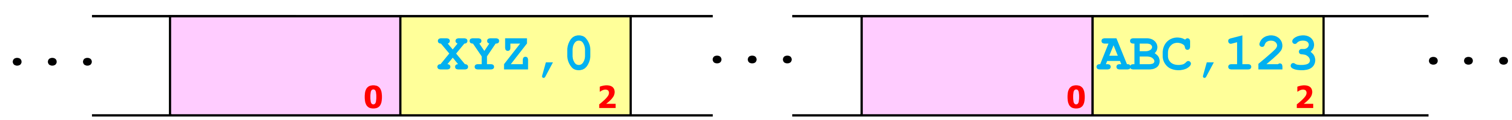
Use #deleted + #data to compute load factor

```
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 [ ]

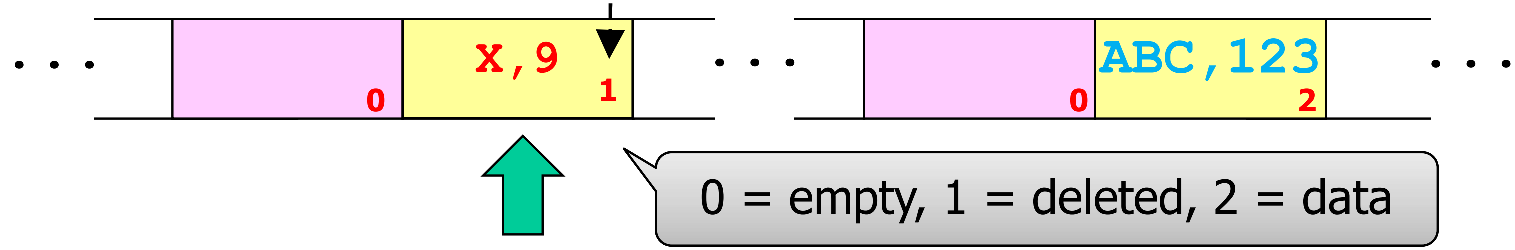
```
MappedT& operator[](const KeyT& key) {
  size_t pos = find_position(key);
  if (mBuckets[pos].available()) { // No data
    insert_to_position(make_pair(key, MappedT()),pos);
  }
  return mBuckets[pos].value.second;
}
```

```
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
```



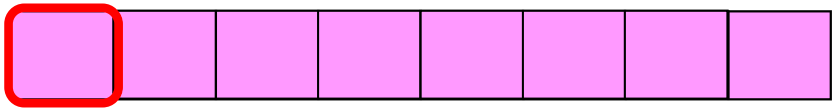
erase

```
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;
  }
}
```



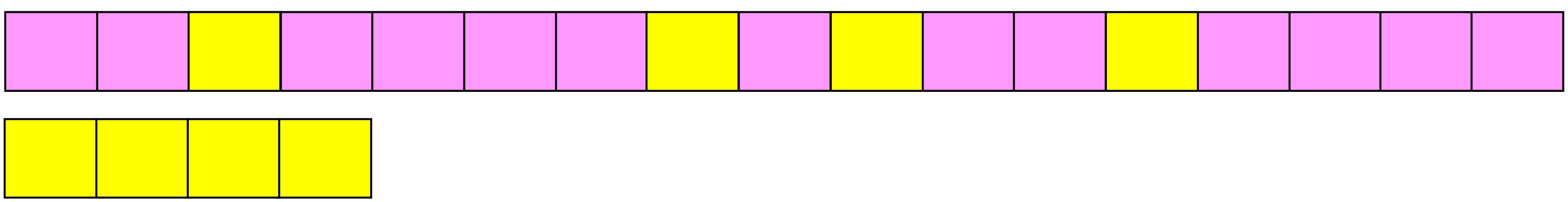
clear

```
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);
  }
}
```



Other functions

```
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)

- When use linear probing and add new data, what’s the most likely location of the new data?

Cookie Monster Effect

การตรวจกำลังสอง (Quadratic Probing)

- To remove primary clustering
- Avoid checking adjacent slots
- Jump further and further

+1, +3, +5, +7,…

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

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

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

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 = mBuskets.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;
}
```

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

Class for computing the next entry to probe

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

Class for computing the next entry to probe

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

```
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;
  }
};
```

```
QuadraticProbing mNextAddress;
...
size_t find_position(const KeyT& key) {
  size_t homePos = hash_to_bucket(key);
  size_t pos = homePos, m = mBuskets.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

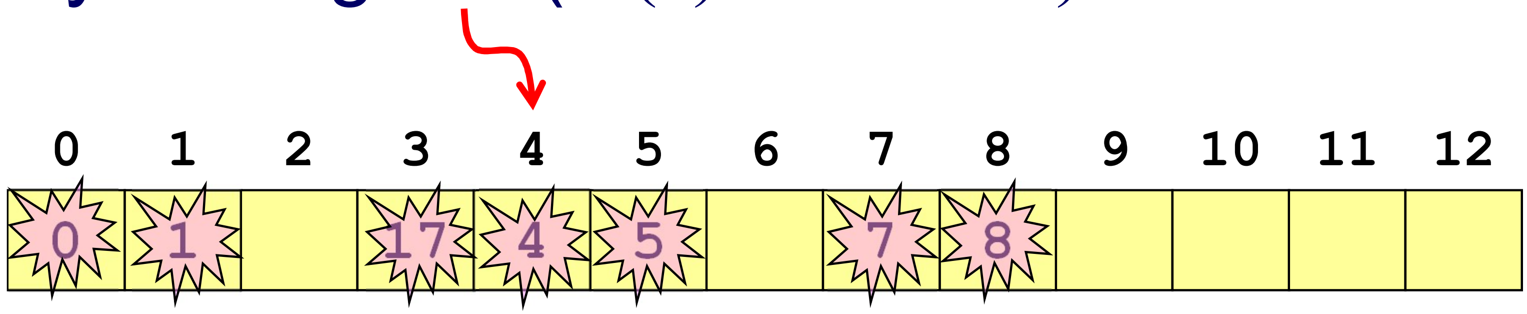
```
class unordered_map {
  ...
  vector<BucketT>  mBuckets;
  size_t           mSize;
  HasherT          mHasher;
  EqualT           mEqual;
  float            mMaxLoadFactor;
  size_t           mUsed;
  NextAddressT     mNextAddress;

  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 )

  { }
```

Quadratic probing does not check every entry!

- Try adding 30 ($h(x) = x \ \% \ 13$)



$h(x) = 4$

$(4+1^2)\%13 = 5$

$(4+2^2)\%13 = 8$

$(4+3^2)\%13 = 0$

$(4+4^2)\%13 = 7$

$(4+5^2)\%13 = 3$

$(4+6^2)\%13 = 1$

$(4+7^2)\%13 = 1$

$(4+8^2)\%13 = 3$

$(4+9^2)\%13 = 7$

$(4+10^2)\%13 = 0$

$(4+11^2)\%13 = 8$

$(4+12^2)\%13 = 5$

$(4+13^2)\%13 = 4$

...

May not find an empty slot,

even though there are many!

When table size is a prime number

- Will check at least half of the entries!
- So, if load factor $\leq$ ½ can guarantee to find empty slot when new data is added!
- Proof : let $0 \leq i < j \leq \lfloor m/2 \rfloor$ if above is not true, there exist the i[th] and the j[th] probe that look at the same location

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

- Impossible : $(j - i)$ not 0, $(j+i)$ not m and $(j - i)(j+i) \ \% \ m \neq 0$ because both (j-i) and (j+i) $< m$ and m is prime

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

If $\lambda_{max} = 0.5$ can guarantee **find_position** will find slot

Clustering

- การเกาะกลุ่มปฐมภูมิ (primary clustering)
  - Can easily see, data is adjacent to each other
  - The bigger the cluster, the faster it grows
  - Search will be slow, like a linear search
- การเกาะกลุ่มทุติยภูมิ (secondary clustering)
  - Data with same $h(x)$ will probe in the same sequence
  - Probing will cost more if there's more collision
  - $h_j(x) = (h(x) + j) \% m, \;\; h_j(x) = (h(x) + j^2) \% m$
  - Can fix this by allowing data with same $h(x)$ to not probe in the same manner
  - The amount to jump should depend on x

การแฮชสองชั้น (Double Hashing)

- Use another hash function to compute how far to jump
- So data that hash to the same entry can probe differently

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

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

- Must ensure $g(x) \% m \neq 0$ (to make progress)
  - $g(x) = R - (x \% R)$ R is prime and $R < m$
- and gcd($g(x)$, m) must == 1
  so as to check every entries!
  - Can guarantee this by ensuring that m is prime!
  - $h(x) = 0$, $g(x) = 4$, $m = 8$ will only check 0 and 4
  - $h(x) = 0$, $g(x) = 4$, $m = 7$ will check 0, 4, 1, 5, 2, 6, 3

Comparing average cost for probing

- Linear probing takes more time
- Quadratic probing and double hashing roughly the same
- If $\lambda \leq 0.5$, not much difference!

	Linear Probing			Quadratic Probing			Double Hashing	
Found?	Yes	No		Yes	No		Yes	No
$\lambda = 0.3$	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

Comparing average number of probe

	Number of probe	
	Found	Not Found
Separate Chaining $(\lambda \geq 0)$	$1 + \lambda/2$	$1 + \lambda$
Linear Probing $(0 \leq \lambda \leq 1)$	$\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$	$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$
Double Hashing $(0 \leq \lambda \leq 1)$	$\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$	$\frac{1}{1-\lambda}$

Q: When use linear probing, if we want the average number of probe to be no more than 5, how large can λ be?

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

Time comparison(java)

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

```
public static void main(String[] args) {
   Set set = new ArraySet();
```

ArraySet,

Set with	Time (ms)
ArraySet	164987
BSTSet	1112
AVLSet	430
LinearProbingHashSet	1903
QuadraticProbingHashSet	390
SeparateChainingHashSet	350

```
}
```

When done set has 73816 data

Points to watch out

- Not good when
  - Go through data with iterator
  - Need order of data, getMin, getMax, ...
  - Will need to search the whole table $\Theta(m+n)$
- Need to ensure h(x) is good
  - If h(x) is not good, will work correctly but can be $O(n)$

```
class BookHasher {
public:
    size_t operator()(const Book& b) const {

          return 0;

    }
};
```

Summary

- Search add remove data in hash table is fast
- Can improve running time by using more space, keeping λ low
- Hash function affects running time

Try to do

- iterator find (const KeyT& key);
- In separate chaining if want to change BucketT from

    typedef vector<ValueT> BucketT;   to

    typedef set<ValueT> BucketT;

  ,what would need to be changed?