Topic 12

Cache vs. hash

資料結構與程式設計

Data Structure and Programming

12/12/2018

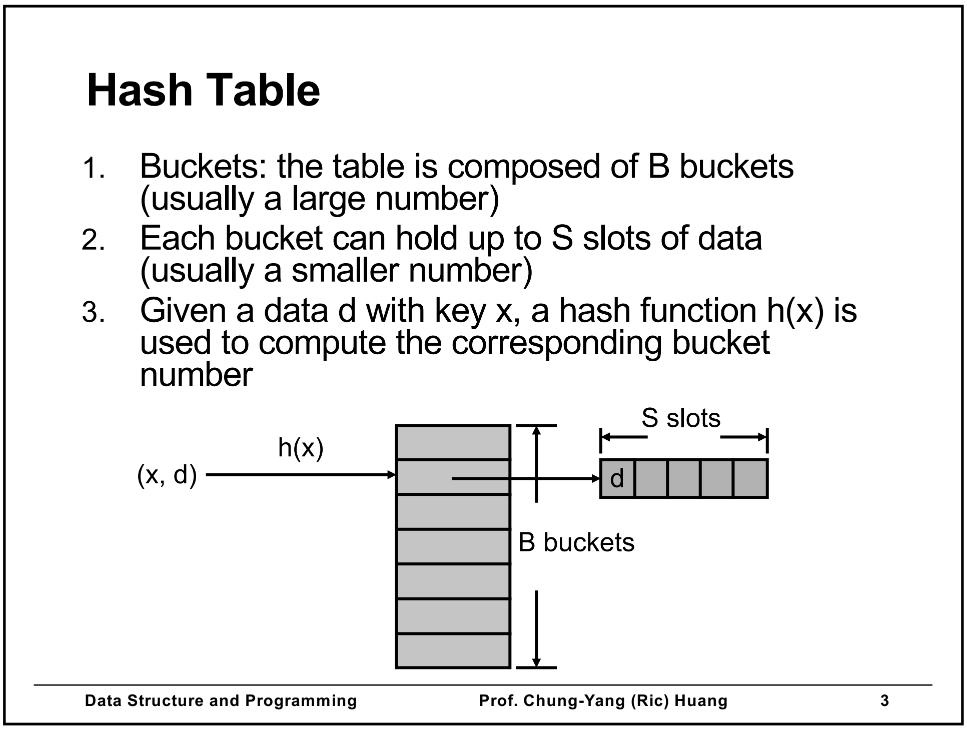
From O(log n) to O(1) ?

- For set and map, they have good complexity for "insert", "delete" and "find" operations

  ➔ O(log n)

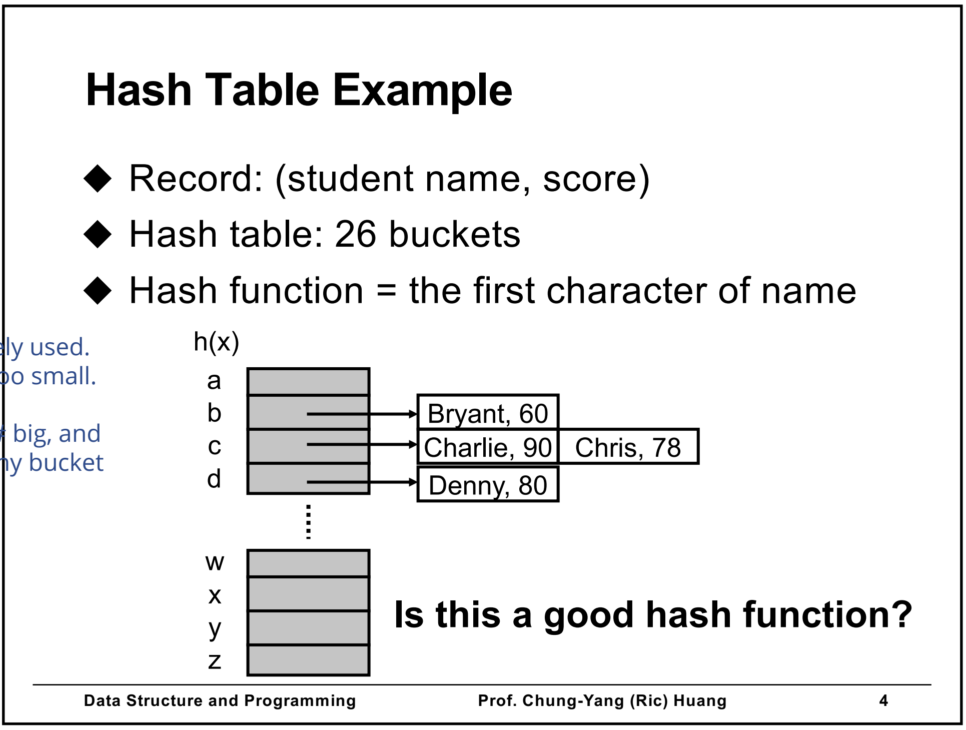
- However, in set and map, all the data are sorted --
  - Can output the data in ascending/descending order
  - Can get the list of elements with values in certain range

- What if we don't care about the order, and just want to have fast "insert", "delete" and "find" operations?
  - Can we gain something (complexity) back for not sorting the data?



Hash Table

1. Buckets: the table is composed of B buckets (usually a large number)
2. Each bucket can hold up to S slots of data (usually a smaller number)
3. Given a data d with key x, a hash function h(x) is used to compute the corresponding bucket number



Hash Table Example

- Record: (student name, score)
- Hash table: 26 buckets
- Hash function = the first character of name

Is this a good hash function?

1. some buckets are rarely used.
2. total bucket count is too small.

that is, we want bucket # big, and possibility of choosing any bucket shall be the same.

Complexity Analysis

- ◆ Depending on how the s slots are designed
- ◆ However, the worse case...
  - Insert: O(1)
    - Assuming it takes O(1) to compute h(x)
  - Delete: O(s) → can they be O(log s)?
  - Find: O(s) → what's the price to pay?
- ◆ Because s is usually a smaller number (e.g. 2) ➔ Very efficient

Hash Table Design Issues

1. Choice of hash function
2. Overflow handling methods
3. Size (number of buckets) of hash table

Hash Functions

- Convert key x to an integer b as the bucket index ($0 \le b \le B - 1$)
  - Generally O(1) complexity
- Discuss: how’s the hash function used in slide #4 (student name, score) example?
  - No good, the first characters of names are usually not evenly distributed among 26 letters
  - Some buckets may become full easily (e.g. ‘c’), while some may be empty (e.g. ‘x’)
- Ideal: for all possible key values, approximately the same number of keys get mapped into each bucket

  ➔ Uniform Hash Function

Hash Function Methods

- Summation
  - e.g. Adding the ASCII values of some/all the characters together
- Shift
  - e.g. Keys are pointers; because pointer values are multiplier of 4 (or 8)

    ➔ h(x) = (x >> 2) ...
  - e.g. x.p, x.q are two pointers;

    ➔ h(x) = (x.p >> 3) + (x.q >> 6);
- Division
  - e.g. Divide a prime number
- Others: folding, mid-square, digit analysis, etc
- Usually: mixed of the above

usually we have bucket # prime number.
folks say that take modulus by prime number would have better, more uniform distribution.

Collision and Overflow

- Collision
  - Two non-identical keys are hashed into the same bucket
  - At most (s – 1) collisions for a buckets
  - Reduced by better hash function
- Overflow
  - When a new key is hashed into a full bucket

➔ For better hash performance,
we should try to produce less collisions and prevent overflow

Overflow Handling

- Overflow may still happen when more and more data are stored into the hash

1. Open addressing
   - Find a non-full bucket to insert the new key
   - Linear probing, quadratic probling, rehashing, (pseudo)random probing, etc
2. Chaining
   - Use linked list, dynamic array, or other kinds of ADT to make the s extendible

Hash Table Size

- ◆ Hash table size (number of buckets) also affects the occurrence of overflow
  - Too small ➔ Overflow happens
  - Too large ➔ Waste of memory

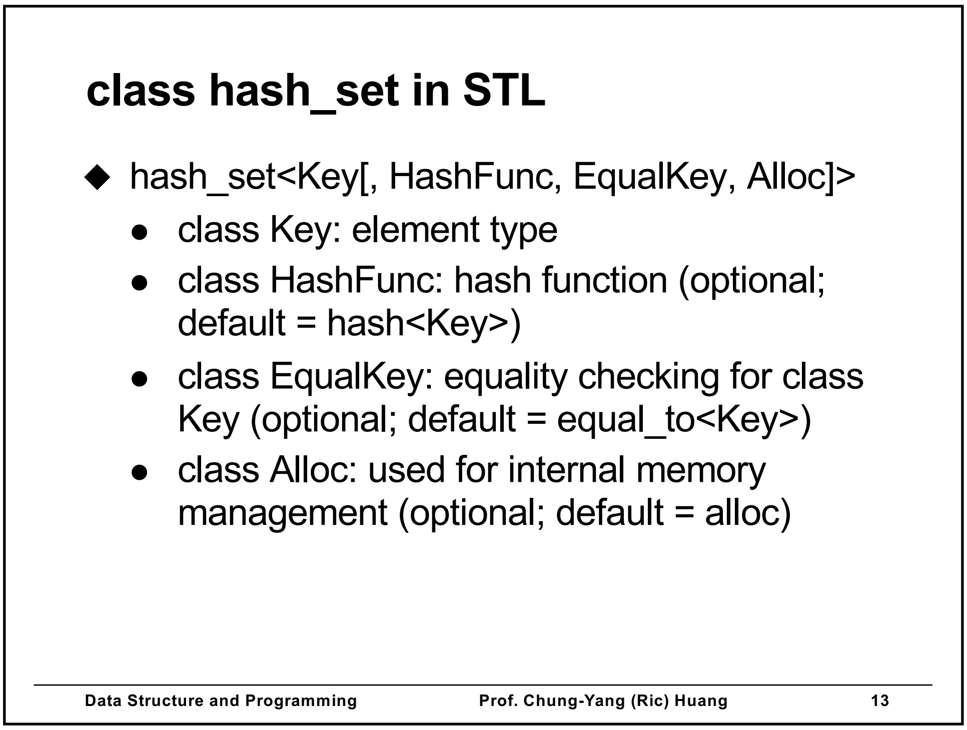
1. Static hashing
   - Fixed-size hash table
   - Easier to implement; better if the number of possible elements is known in advance
2. Dynamic hashing
   - Hash table size can grow when necessary
   - When it grows, rehashing is needed

Hash Classes in STL

- ◆ Static hashing, use set or map for the storage of each bucket

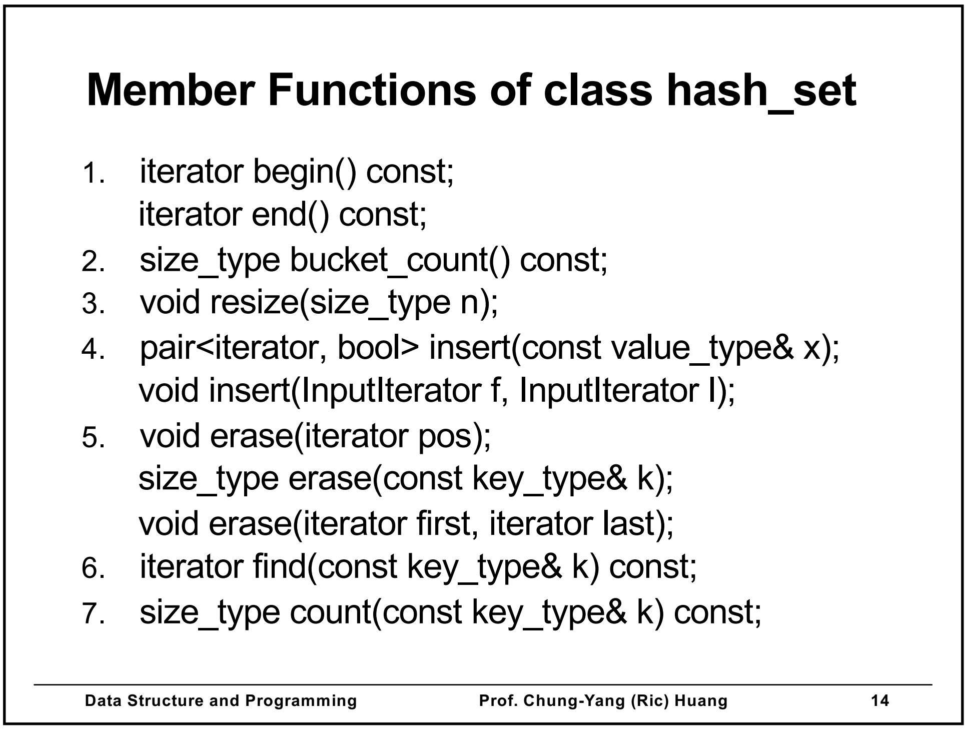
1. hash_set
2. hash_multiset
3. hash_map
4. hash_multimap

- ◆ However, hash is NOT included as a standard package in newer platforms. You may need to do (For example) ---
  - #include <hash_set.h> and/or
  - g++ -I/usr/include/c++/4.0.0/backward/



class hash_set in STL

- hash_set<Key[, HashFunc, EqualKey, Alloc]>
  - class Key: element type
  - class HashFunc: hash function (optional; default = hash<Key>)
  - class EqualKey: equality checking for class Key (optional; default = equal_to<Key>)
  - class Alloc: used for internal memory management (optional; default = alloc)



Member Functions of class hash_set

1. iterator begin() const;
   iterator end() const;
2. size_type bucket_count() const;
3. void resize(size_type n);
4. pair<iterator, bool> insert(const value_type& x);
   void insert(InputIterator f, InputIterator l);
5. void erase(iterator pos);
   size_type erase(const key_type& k);
   void erase(iterator first, iterator last);
6. iterator find(const key_type& k) const;
7. size_type count(const key_type& k) const;

Other Hash Classes in STL

- class hash_multiset in STL
  - Similar to hash_set, but allow elements with identical keys
- class hash_map
  - hash_map<Key, Data[, HashFunc, EqualKey, Alloc]>
- class hash_multimap
  - Similar to hash_map, but allow elements with identical keys

Hash Implementation Example (myHashSet.h)

```
template <class Data>
class HashSet
{
public:
   class iterator
   {
      friend class HashSet<Data>;
   };
private:
   size_t                      _numBuckets;
   // Each _buckets[i] is a vector<Data>
   vector<Data>*               _buckets;
};
```

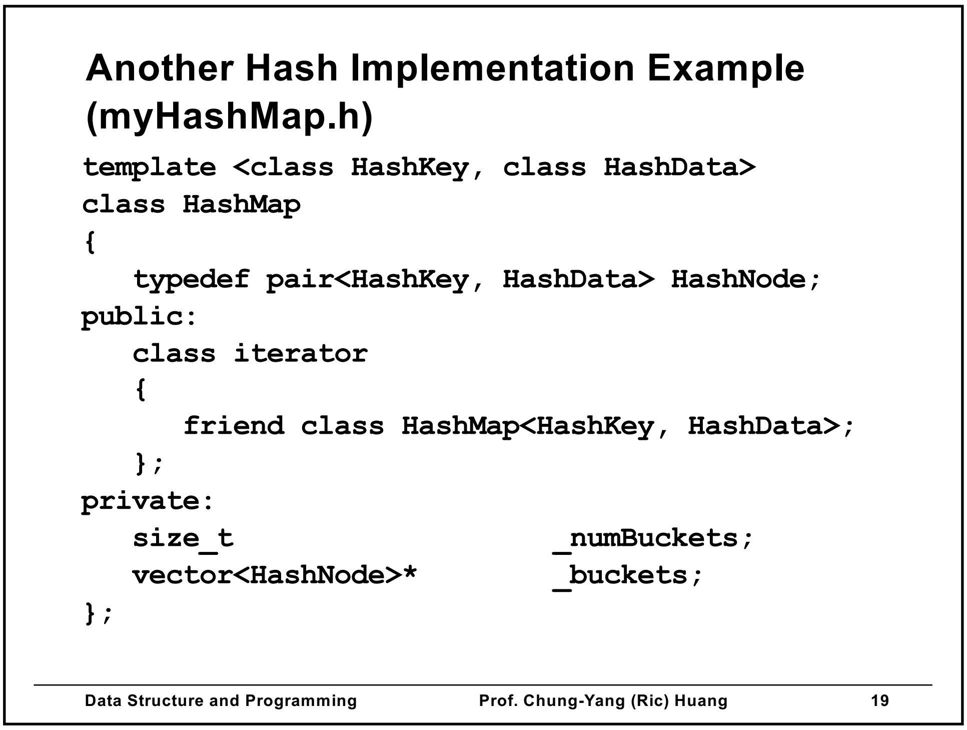
Must overload:
- operator () → as the hash function (i.e. d() % _numBuckets to return bucket index)
- operator == → to compare if the data are equivalent (or have the same key)

Supported functions in class HashSet

```
iterator begin() const; // Point to the first valid data
iterator end() const; // Past the end
bool empty() const; // return true if no valid data
size_t size() const; // number of valid data
vector<Data>& operator [] (size_t i) { return _buckets[i]; }
const vector<Data>& operator [](size_t i) const;
void init(size_t b); // initialize Hash with _numBuckets = b
void reset();
```

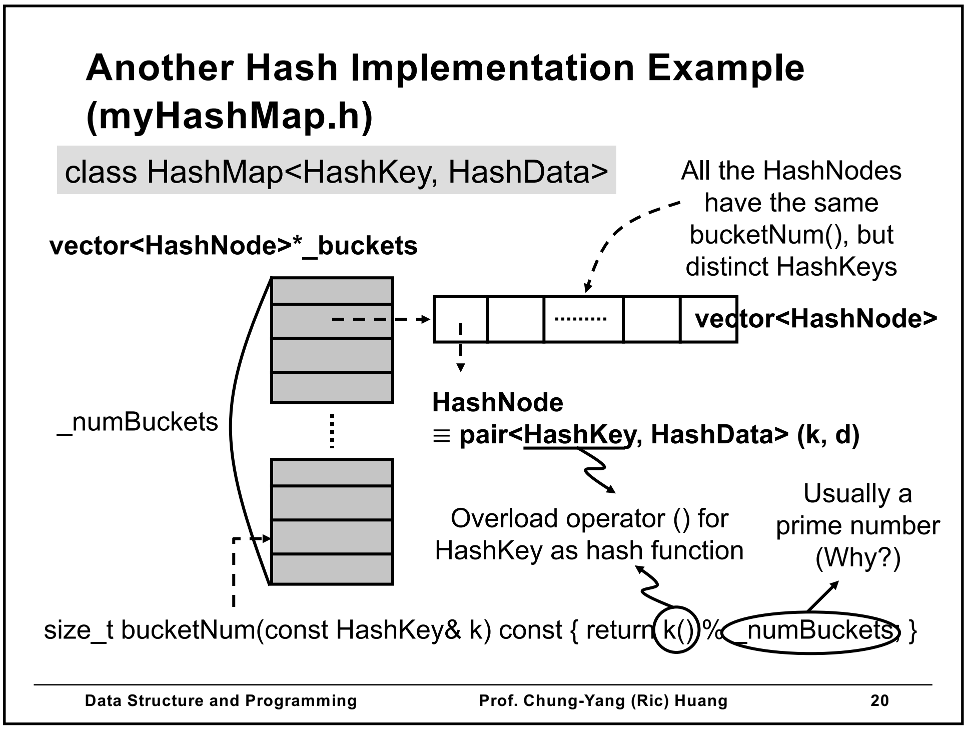
Supported functions in class HashSet

```
// check if d is in the hash...
// if yes, return true;  else return false;
bool check(const Data& d) const;

// query if d is in the hash...
// if yes, replace d with the data in the hash and return true;
// else return false;
bool query(Data& d) const;

// update the entry in hash that is equal to d (i.e. == return true)
// if found, update that entry with d and return true;
// else insert d into hash as a new entry and return false;
bool update(const Data& d);

// return true if inserted successfully (i.e. d is not in the hash)
// return false is d is already in the hash ==> will not insert
bool insert(const Data& d);
```



Another Hash Implementation Example (myHashMap.h)

```
template <class HashKey, class HashData>
class HashMap
{
   typedef pair<HashKey, HashData> HashNode;
public:
   class iterator
   {
      friend class HashMap<HashKey, HashData>;
   };
private:
   size_t                    _numBuckets;
   vector<HashNode>*         _buckets;
};
```

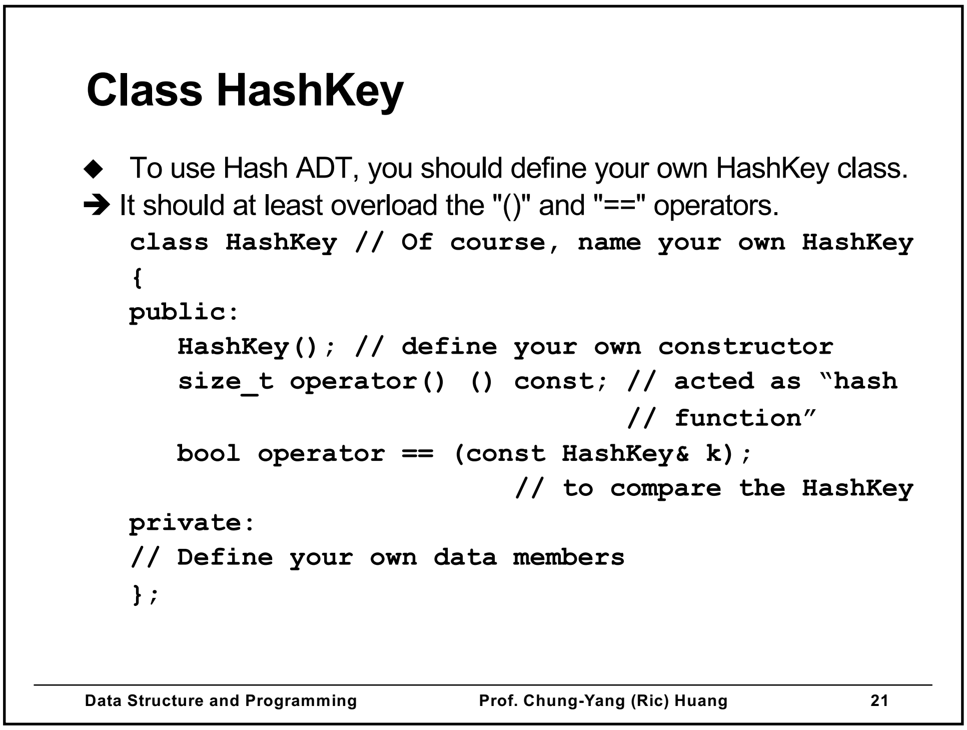


Another Hash Implementation Example (myHashMap.h)

class HashMap<HashKey, HashData>

vector<HashNode>*_buckets
All the HashNodes have the same bucketNum(), but distinct HashKeys
....
vector<HashNode>
_numBuckets
HashNode
≡ pair<HashKey, HashData> (k, d)
Overload operator () for HashKey as hash function
Usually a prime number (Why?)

size_t bucketNum(const HashKey& k) const { return k() % _numBuckets; }

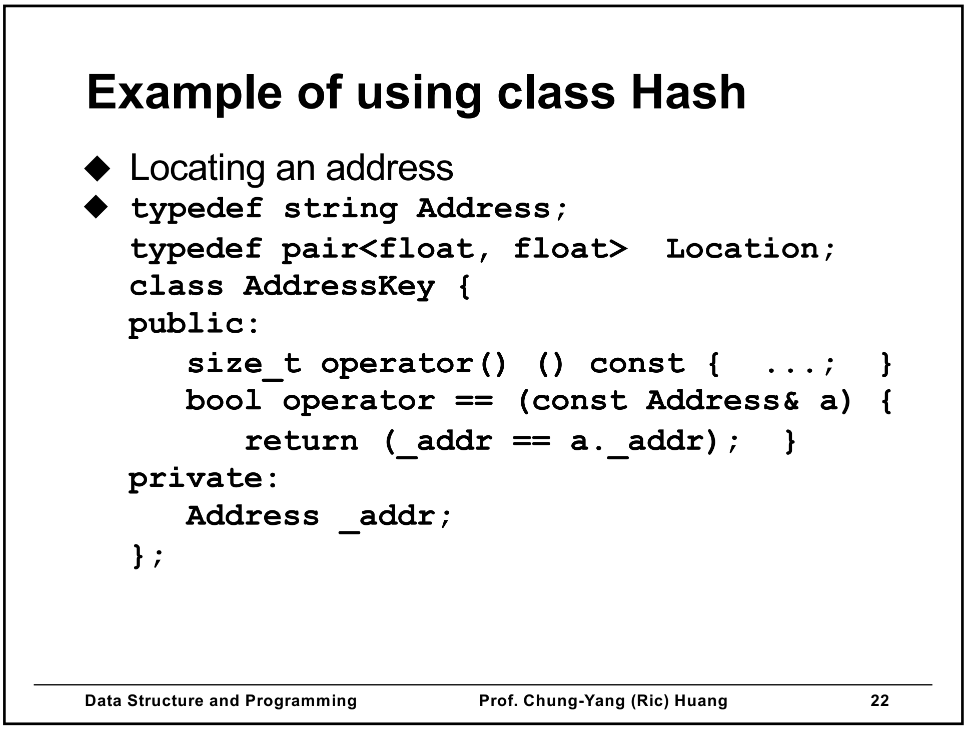


Class HashKey

- To use Hash ADT, you should define your own HashKey class.

➔ It should at least overload the "()" and "==" operators.

```
class HashKey // Of course, name your own HashKey
{
public:
   HashKey(); // define your own constructor
   size_t operator() () const; // acted as “hash
                                // function”
   bool operator == (const HashKey& k);
                         // to compare the HashKey
private:
// Define your own data members
};
```



Example of using class Hash

- Locating an address
- `typedef string Address;`

```
typedef pair<float, float>  Location;
class AddressKey {
public:
   size_t operator() () const {  ...;  }
   bool operator == (const Address& a) {
      return (_addr == a._addr);  }
private:
   Address _addr;
};
```

class Hash::iterator

```
class iterator
{
   friend class HashMap<HashKey, HashData>;
private:
// Define your own data members!!
};
```

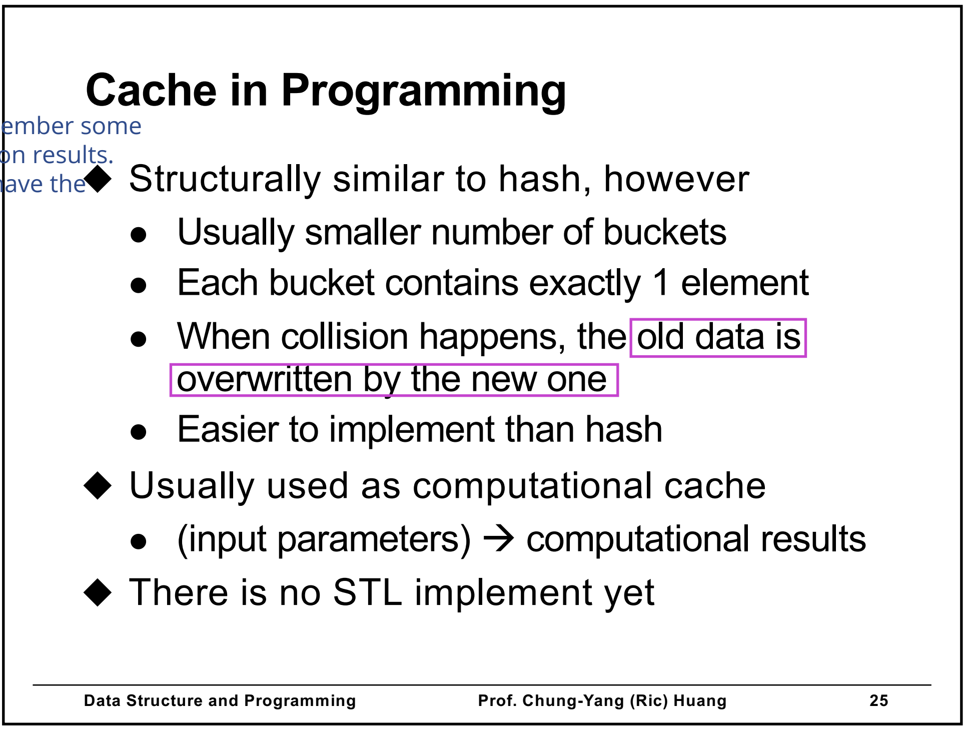
- ◆ Purpose: to go through the “valid” HashNodes in the Hash
- ◆ To use:

```
HashMap<HashKey, HashData> hh;
... // insert data
HashMap<HashKey, HashData>::iterator hi = hh.begin();
for (; hi != hh.end(); ++hi)
   cout << (*hi).first << “ → ” << (*hi).second
        << endl;
```

Supported functions in class HashMap

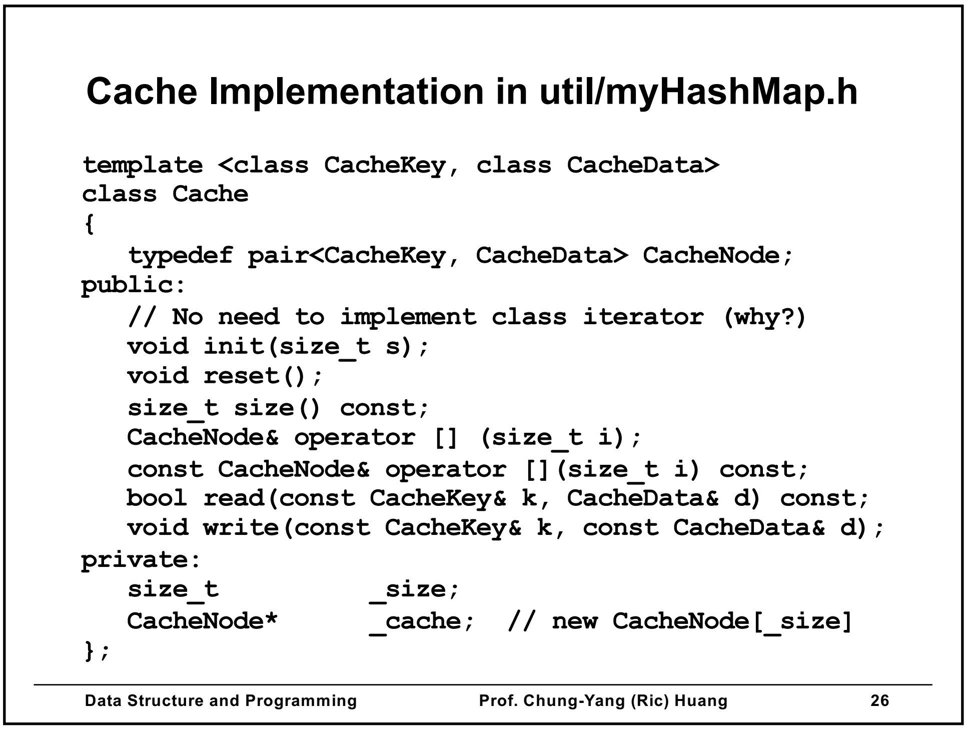
```
iterator begin() const; // Point to the first valid data
iterator end() const; // Pass the end
bool empty() const; // return true if no valid data
size_t size() const; // number of valid data
vector<HashNode>& operator [] (size_t i) { return _buckets[i]; }
const vector<HashNode>& operator [](size_t i) const;
void init(size_t b); // initialize Hash with _numBuckets = b
void reset();
bool check(const HashKey& k, HashData& n);
bool insert(const HashKey& k, const HashData& d);
bool replaceInsert(const HashKey& k, const HashData& d);
void forceInsert(const HashKey& k, HashData d);
```

use a data structure to remember some time-consuming computation results. it's like cache in CPU, thus have the name.



Cache in Programming

- ◆ Structurally similar to hash, however
  - Usually smaller number of buckets
  - Each bucket contains exactly 1 element
  - When collision happens, the old data is overwritten by the new one
  - Easier to implement than hash
- ◆ Usually used as computational cache
  - (input parameters) → computational results
- ◆ There is no STL implement yet



Cache Implementation in util/myHashMap.h

```
template <class CacheKey, class CacheData>
class Cache
{
   typedef pair<CacheKey, CacheData> CacheNode;
public:
   // No need to implement class iterator (why?)
   void init(size_t s);
   void reset();
   size_t size() const;
   CacheNode& operator [] (size_t i);
   const CacheNode& operator [](size_t i) const;
   bool read(const CacheKey& k, CacheData& d) const;
   void write(const CacheKey& k, const CacheData& d);
private:
   size_t          _size;
   CacheNode*      _cache;  // new CacheNode[_size]
};
```

Example of using class Cache (BDD project)

◆ Computed table

ITE(F, G, H) = F * G + $\overline{F}$ * H

F, G, H: BddNode (contains a size_t data)

➔ Requires expensive recursive calls to compute ITE() functions (O(|F|*|G|*|H|))

➔ The computed table (cache) is to record the result (as CacheData) with respect to the ITE parameters (as CacheKey)

➔ So next time when the same ITE(F, G, H) computation is required, we can immediately look up the cached result

FYI: Google Hash

http://code.google.com/p/google-sparsehash/

◆Two hashtable implementations:

1. Sparse hash: designed to be space efficient
2. Dense hash: designed to be time efficient

◆ Interface very similar to SGI’s (STL) hash_map, hash_set, etc. But is claimed to be much more efficient.

- sparse_hash_map, sparse_hash_set, dense_hasp_map, dense_hash_set
- e.g. sparse_hash_map
  <Key, Data, HashFcn, EqualKey, Alloc>