CSE221 Data Structures
Lecture 16: Maps and Hashing

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

November 10, 2021

Introduction

- I updated attendance records. They can be found in the portal under E-attendance.
- I will post Assignment 3 this week.
- Reference for this lecture: Textbook Chapter 9.1–9.2.

Maps

- A *map* is an ADT that stores elements so they can be located quickly using keys.
- Specifically, a map stores key-value pairs (k, v), which we call *entries*, where k is the key and v is its corresponding value.

Example

We could store student records in a map data structure. The key k would be the student ID number, and v is the student's records. Then the records of a student can be quickly accessed by looking up his ID number.

- Keys are *unique*. In the example above, for instance, no two students can have the same ID number.

Entries

```
template <typename K, typename V>
class Entry {                          // a (key, value) pair
public:                                 // public functions
  Entry(const K& k = K(), const V& v = V())
    : _key(k), _value(v) { }                // constructor
  const K& key() const { return _key; }        // get key
  const V& value() const { return _value; }  // get value
  void setKey(const K& k) { _key = k; }        // set key
  void setValue(const V& v) { _value = v; }  // set value
private:                                    // private data
  K _key;                                            // key
  V _value;                                        // value
};
```

- This is an example of an object-oriented design pattern, the *composition pattern*, which defines a single object that is composed of other objects.

The Map ADT

Operation	Output	Map
empty()	TRUE	$\emptyset$
put(5, A)	p_1 : [(5, A)]	$\{(5, A)\}$
put(7, B)	p_2 : [(7, B)]	$\{(5, A), (7, B)\}$
put(2,C)	p_3 : [(2,C)]	$\{(5, A), (7, B), (2, C)\}$
put(2, E)	p_3 : [(2, E)]	$\{(5, A), (7, B), (2, E)\}$
find(7)	p_2 : [(7, B)]	$\{(5, A), (7, B), (2, E)\}$
find(4)	end	$\{(5, A), (7, B), (2, E)\}$
find(2)	p_3 : [(2, E)]	$\{(5, A), (7, B), (2, E)\}$
size()	3	$\{(5, A), (7, B), (2, E)\}$
erase(5)	-	$\{(7, B), (2, E)\}$
erase(p_3)	-	$\{(7, B)\}$
find(2)	end	$\{(7, B)\}$

The Map ADT

- **size**(): Return the number of entries in M.
- **empty**(): Return true if M is empty and false otherwise.
- **find**(k): If M contains an entry $e = (k, v)$, with key equal to k, then return an iterator p referring to this entry, and otherwise return the special iterator `end`.
- **put**(k, v): If M does not have an entry with key equal to k, then add entry (k, v) to M, and otherwise, replace the value field of this entry with v; return an iterator to the inserted/modified entry.
- **erase**(k): Remove from M the entry with key equal to k; an error condition occurs if M has no such entry.
- **erase**(p): Remove from M the entry referenced by iterator p; an error condition occurs if p points to the end sentinel.
- **begin**(): Return an iterator to the first entry of M.
- **end**(): Return an iterator to a position just beyond the end of M.

C++ Interface

```
template <typename K, typename V>
class Map {                                          // map interface
public:
  class Entry;                                  // a (key,value) pair
  class Iterator;                       // an iterator (and position)
  int size() const;                   // number of entries in the map
  bool empty() const;                           // is the map empty?
  Iterator find(const K& k) const; //find entry with key k
  Iterator put(const K& k, const V& v); // insert/replace
  void erase(const K& k);             // remove entry with key k
  void erase(const Iterator& p);               // erase entry at p
  Iterator begin();                     // iterator to first entry
  Iterator end();                         // iterator to end entry
};
```

Maps

- The interface above uses *iterators*. (See Lecture 11.)
- Operators *, $++$, $--$ and $==$ are overloaded in the same way.
- In particular, * returns a reference to the associated entry.

- A map can be implemented using a doubly-linked list.
- Problem: the main operations `find`, `put` and `erase` take $O(n)$ time.

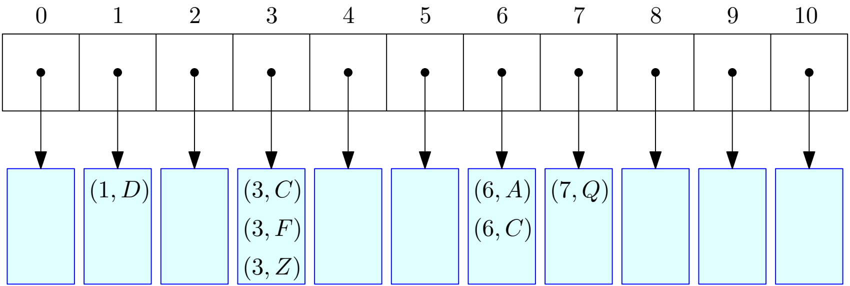
- STL provides a map data structure.

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
```

STL Maps

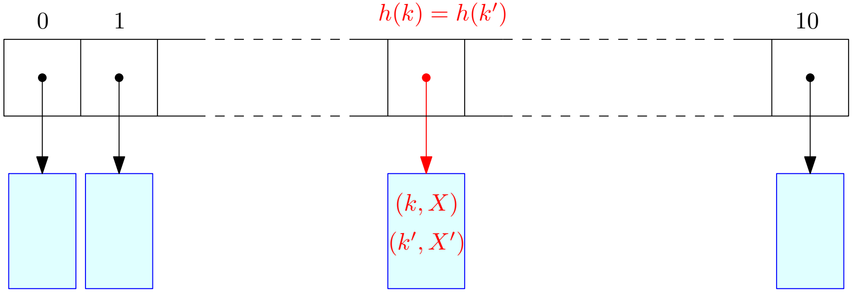
```
map<string, int> myMap;                     // a (string,int) map
map<string, int>::iterator p;   // an iterator to the map
myMap.insert(pair<string, int>("Rob", 28));
myMap["Joe"] = 38;                          // insert("Joe",38)
myMap["Joe"] = 50;                      // change to ("Joe",50)
myMap["Sue"] = 75;                          // insert("Sue",75)
p = myMap.find("Joe");                       // *p = ("Joe",50)
myMap.erase(p);                            // remove ("Joe",50)
myMap.erase("Sue");                        // remove ("Sue",75)
p = myMap.find("Joe");
if (p == myMap.end())
  cout << "nonexistent\n";        // outputs: "nonexistent"
for (p = myMap.begin(); p != myMap.end(); ++p) {
  cout << "(" << p->first << "," << p->second << ")\n";
}                                          // print all entries
```

Bucket Arrays



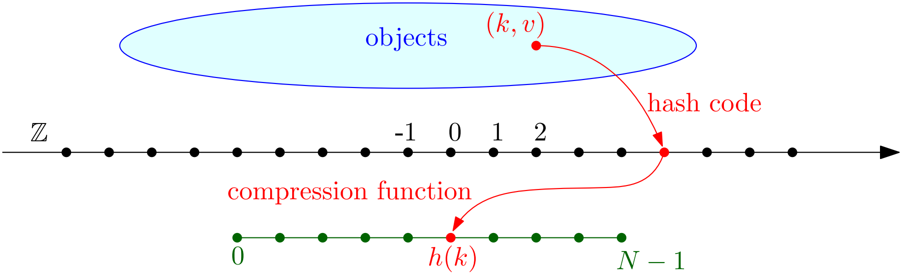
- A *bucket array* is an array A of size N, where each cell of A is thought of as a "bucket" (that is, a collection of key-value pairs) and the integer N defines the capacity of the array.
- If the keys are well-distributed in $[0 \ldots N-1]$ and n is not much smaller than N, then the map ADT operations can be done in $O(1)$ time with a bucket array.

Hash Tables



- If the keys do not satisfy the conditions above, we use a *hash function* $h : K \to [0 \ldots N-1]$ that map the keys to $[0 \ldots N-1]$. Then (k, X) is stored in the bucket with index $h(k)$.
- Collisions can occur: if two keys k, k' have same hash value $h(k) = h(k')$, then two pairs (k, X) and (k', X') may be stored in the same bucket.

Hash Functions



- If there are not too many collisions, this data structure will be very efficient. So we need to design good hash functions in the sense that they minimize the number collisions. Intuitively, we want $h(k)$ to behave like a random number, so as to minimize collisions.
- To compute $h(k)$, we first compute an integer called *hash code*, which we map to $\{0, 1, \ldots, N-1\}$ using a compression function.

Converting to an Integer

- If k is of type `char` or `short`, we can simply cast k to an `int`.
- If k is of type `long`, we can just keep the last bits.
- For instance, if `long` takes 64 bits and `int` takes 32, we map x to $x \bmod 2^{32}$. So if $x = 2^{32}a + b$, where a, b are 32-bit integers, then x is mapped to b.
- Problem: It ignores a. Instead, we can map x to $a + b$.
- More generally, any object can be represented as a ℓ-tuple of integers $(x_0, x_1, \dots, x_{\ell-1})$.
- Then we can map x to $\sum_{i=0}^{\ell-1} x_i$.

Example

A floating point number $x = x_0.10^{x_1}$ where x_0, x_1 are of type `int` can be mapped to $x_0 + x_1$.

Converting to an Integer

- Suppose we apply the method above for strings.
- For instance, the string "$a_0 a_1 \dots a_{\ell-1}$" is converted to $x_0 + x_1 + \cdots + x_{\ell-1}$ where x_i is the ASCII code of x_i.
- Then `"temp01"` and `"temp10"` are mapped to the same integer.
- Similarly, `"stop"`, `"pots"` and `"tops"` are mapped to the same integer.
- So there are a lot of collisions.

Polynomial Hash Codes

- To fix this problem, we can use a *polynomial hash code*:

$$(x_0, \ldots, x_{\ell-1}) \mapsto x_0 a^{\ell-1} + \cdots + x_{\ell-3} a^2 + x_{\ell-2} a + x_{\ell-1}$$

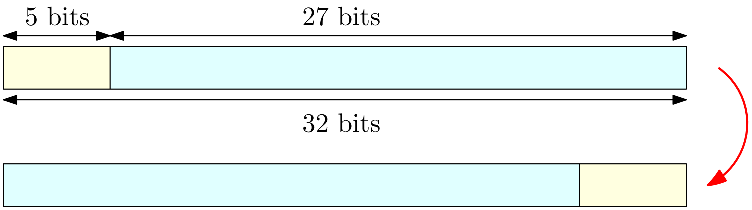
  where a is a constant integer, $a \neq 0$ and $a \neq 1$.

- It can be computed in $O(\ell)$ time by *Hörner's rule*:

$$x_{\ell-1} + a(x_{\ell-2} + a(x_{\ell-3} + \ldots a(x_1 + ax_0)) \ldots)$$

- In this calculation, the results will overflow the maximum size of integers (i.e. intermediate results will be greater than 2^{32}) but we can ignore it.
- Experiments suggest that for English words, the best choices are $a = 33$, 37, 39 and 41.

Cyclic Shift Hash Codes




```
int hashCode(const char* p, int len) {
  unsigned int h = 0;                  // hash a character array
  for (int i = 0; i < len; i++) {
    h = (h << 5) | (h >> 27);           // 5-bit cyclic shift
    h += (unsigned int) p[i];          // add in next character
  }
  return hashCode(int(h));
}
```

- Experimentally, for English words, a shift of 5 gives the best results.

Hashing Floating-Point Quantities

```
int hashCode(const float& x) {
  int len = sizeof(x);
  const char* p = reinterpret_cast<const char*>(&x);
  return hashCode(p, len);
}
```

- Here, `reinterpret_cast` converts the floating point number x into an array of characters.
- This is not done in a meaningful way: It just reads the internal bit representation of x.

Compression Functions

- One simple *compression function* to use is

$$h(k) = |k| \bmod N,$$

  which is called the *division method*.

- N is often chosen to be a prime number. Otherwise, repeated patterns are more likely to generate collisions.

Example

If $N = 100$ and the keys are $\{200, 205, 210, 215, 220, \dots 600\}$ then each hash code collides with at least 3 others.

- Even when N is prime, this problem may happen.

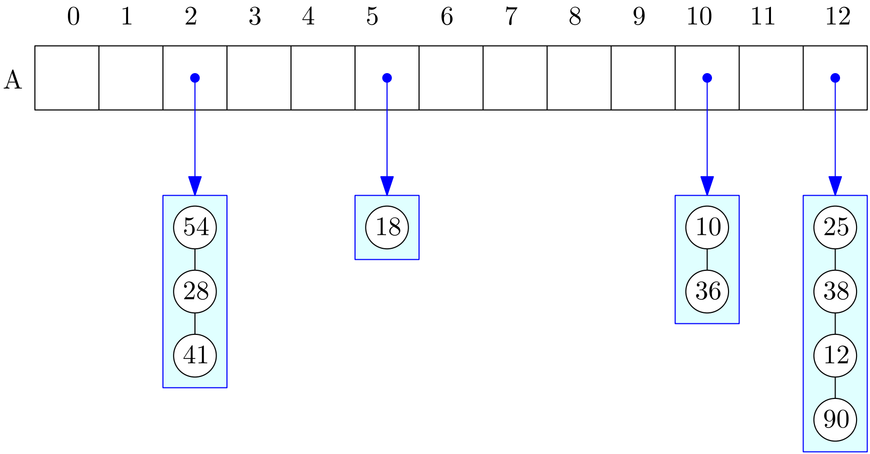
The MAD Method

- The *multiply, add and divide* (or "MAD") method helps eliminate such patterns:

$$h(k) = |ak + b| \bmod N,$$

  where N is a prime number, and a and b are nonnegative integers randomly chosen at the time the compression function is determined, so that $a \bmod N \neq 0$.
- This compression function is chosen in order to eliminate repeated patterns in the set of hash codes and to get us closer to having a "good" hash function, that is, one having the probability that any two different keys collide is $1/N$.

Collision-Handling Schemes



- Separate chaining with $h(k) = k \bmod 13$.

Collision-Handling Schemes

- Each bucket $A[i]$ store a small map, M_i, implemented using a list, holding entries (k, v) such that $h(k) = i$.
- So each separate M_i chains together the entries that hash to index i in a linked list.
- This *collision-resolution* rule is known as *separate chaining*.

Pseudocode

procedure $\textsc{Find}(k)$
    return $A[h(k)]$.find(k)

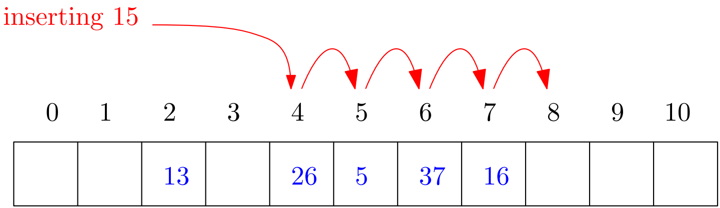
procedure $\textsc{Put}(k, v)$
    $p \leftarrow A[h(k)]$.put(k)
    $n \leftarrow n + 1$
    return p

Collision-Handling Schemes

procedure $\text{PUT}(k)$
    $A[h(k)]$.erase(k)
    $n \leftarrow n - 1$

- Here, the list-based implementation of the map M_i is good enough because we expect $A[i]$ to be small.
- With a good hash function, we expect each bucket bucket to be of size roughly n/N.
- This value $\lambda = n/N$ is called the *load factor* of the hash table.
- It should be bounded by by a small constant, preferably below 1.
- Then expected running time of operations `find`, `put` and `erase` is $O(\lceil n/N \rceil) = O(1)$.

Linear Probing



- In *open-addressing* schemes, we place at most one entry per bucket.
- One such schemes is *linear probing*.
- We try to insert an entry (k, v) into a bucket $A[i]$, where $i = h(k)$.
- If it is already occupied, then we try t $A[(i+1) \bmod N]$.
- If it is also occupied, then we try $A[(i+2) \bmod N] \ldots$

Linear Probing

- To perform an `erase(k)` operation, it looks like we might need to shift a lot of cells.
- A better way to do it is to write a special "available" marker object.
- Then a `find` operation skips over it.

- An alternative to linear probing is *quadratic probing* where we iterate over buckets

$$A[(i + f(j)) \bmod N] \quad \text{for } j = 0, 1, 2, \ldots \text{ where } f(j) = j^2$$

  until finding an empty bucket.
- It avoids some clustering patterns, but in some cases, it may not find an empty bucket even if there is one.

Double Hashing

- In *double-hashing*, we choose a secondary hash function h', and if h maps some key k to a bucket $A[i]$ with $i = h(k)$ that is already occupied, then we iteratively try the buckets

$$A[(i + f(j)) \bmod N] \quad \text{for } j = 1, 2, 3, \ldots$$

  where $f(j) = j \cdot h'(k)$.

- These open-addressing schemes save some space over the separate-chaining method, but they are not necessarily faster. In experimental and theoretical analyses, the chaining method is either competitive or faster than the other methods, depending on the load factor of the bucket array. So, if memory space is not a major issue, the collision-handling method of choice seems to be separate chaining

Load Factors and Rehashing

- In all the schemes above, we should keep the load factor λ below 1.
- Experiments and average-case analysis suggest that we should have $\lambda < 0.5$ for the open-addressing schemes and $\lambda < 0.9$ for separate chaining.
- If λ becomes too high, we should resize the hash table. This is called *rehashing*. For instance, we can double its size.