Hash Tables

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Outline

- Discuss storing unrelated/unordered data
  - IP addresses and domain names

- Consider conversions between these two forms

- Introduce the idea of hashing:
  - Reducing **O**(ln(*n*)) operations to **O**(1)

- Consider some of the weaknesses

Problem: IP Addresses

- Examples:
  - You want to map an IP address to a corresponding domain name

- A 32-bit IP address is often written as four byte values from 0 to 255
  - Consider IP Address
    - $10010011\ 00101111\ 01101010\ 00011010_2$
    - This can be written as 147.47.106.50
  - Suppose its domain name is
    - ece.snu.ac.kr
  - We use domain names because IP addresses are not human readable

Example: IP Addresses

- Given an IP address, if we wanted to quickly find any associated domain name, we could create an array of size 2^{32} (4294967296) of strings:

```
int const MAX_IP_ADDRESSES = 4294967296;
string domain_name[MAX_IP_ADDRESSES];
```

- For example, 147.47.106.50 can be translated
  - As $147 * 256^3 + 47 * 256^2 + 106 * 256 + 50 = 2469358130$,
- domain_name[2469358130] = “ece.snu.ac.kr";
- Can we use much less memory than this?

Goals and Requirements

- Our goal
  - Store data so that all operations are $\Theta(1)$ time

- Requirement
  - The memory requirement should be $\Theta(n)$

- Can we achieve this goal with data structures we covered before?
  - Lists, stack, queue, trees, …

Goals and Requirements

- In general, we would like to:
  - Create an array of size M
  - Store each of n objects in one of the M bins
  - Have some means of determining the bin in which an object is stored

Idea: Grade Table Example

- Let's try a simpler problem
  - How do I store your examination grades so that I can access your grades in $\Theta(1)$ time?

- Observation: SNU ID is an 9-digit number
  - We can't create an array of size 10^9
  - We can create an array of size 1000 though
  - How could you convert an 9-digit number into a 3-digit number?
  - First three digits might cause a problem
    - almost all students start with 2017, 2018, 2019, …
  - The last four digits, however, are (somehow) random

- Therefore, I could store the examination grade for SNU ID 202101011
  - grade[011] = 99;

Idea: Grade Table Example

- Consequently, I have a function, mapping a student ID to a 3-digit number
    - We can store something in that location
    - Storing it, accessing it, and erasing may take Θ(1)
    - Problem: two or more students may map to the same number:
        - Vayne has ID 200703456
        - Teemo has ID 200301456
        - Both would map to 456

⋮	⋮
454	
455	
456	86
457	
458	
459	
460	
461	
462	
463	79
464	
465	
⋮	⋮

Probability of Collision

- Question:
  - What is the likelihood that in a class of size 100 that no two students will have the same last three digits?
  - Not very high:

$$1 \cdot \frac{999}{1000} \cdot \frac{998}{1000} \cdot \frac{997}{1000} \cdot \quad \cdots \quad \cdot \frac{901}{1000} \approx 0.005959$$

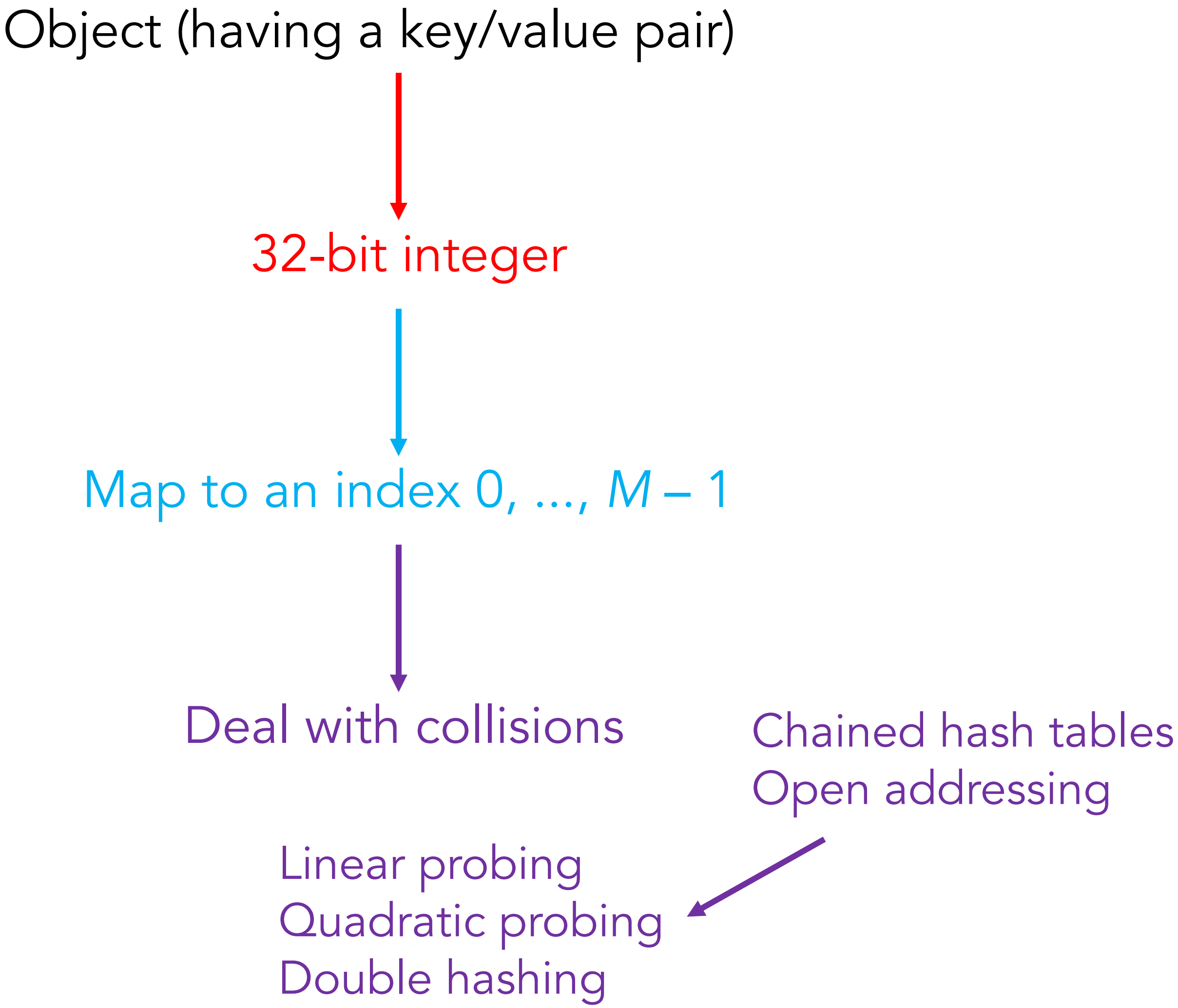
  - Probability of having collision(s): $1 - \ 0.005959 = 0.994041$
  - Implication: If you insert 100 students to the table, there will be at least one collision at the probability of more than 99.4%
    - So highly likely there will be a collision if only using the last three digits

Check the birthday problem: https://en.wikipedia.org/wiki/Birthday_problem

The hashing problem

- The process of mapping an object or a number onto an integer in a given range is called hashing
- Problem: multiple objects may hash to the same value
  - Such an event is termed a collision
- Hash tables use a hash function together with a mechanism for dealing with collisions

The hash process



Object (having a key/value pair)

32-bit integer

Map to an index 0, ..., $M-1$

Deal with collisions

Chained hash tables
Open addressing

Linear probing
Quadratic probing
Double hashing

Summary

- ☐ Discuss storing unordered data
- ☐ Discuss IP addresses and domain names
- ☐ Discussed the issues with collisions

References

[1] Wikipedia, http://en.wikipedia.org/wiki/Hash_table
[2] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.
[3] Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley.

Hash Functions

Kyunghan Lee
Networked Computing Lab (NXC Lab)
Department of Electrical and Computer Engineering
Seoul National University
https://nxc.snu.ac.kr
kyunghanlee@snu.ac.kr

Definitions

- What is a hash of an object?
- From Merriam-Webster:
  - *a restatement of something that is already known*

- The ultimate goal is to map onto an integer range

$$0, 1, 2, \ldots, M - 1$$

The Hash process

Input key

↓

32-bit integer

↓

Map to an index 0, …, $M - 1$

Ideal properties of a hash function

- A hash function is a function mapping an input key to a certain integer range (say 0 to 2^{32} here)

- Necessary properties of such a hash function *h* are:
  - 1a. Computation should be fast, ideally $\Theta(1)$
  - 1b. The hash value must be *deterministic*
    - It must always return the same output
    - If $x = y \Rightarrow h(x) = h(y)$
  - 1c. If two objects are randomly chosen, there should be only a one-in-2^{32} chance that they have the same hash value

Types of hash functions

- Hash functions for different types of input keys
  - General class object
  - Integer
  - String

Hash Function for Class Object

□ The easiest solution is to give each object a unique number

```
class Product {
    private:
        unsigned int hash_value;
        static unsigned int hash_count;
    public:
        Product();
        unsigned int hash() const;
};

unsigned int Product::hash_count = 0;
```

```
Product::Product() {
    hash_value = hash_count;
    ++hash_count;
}

unsigned int Product::hash() const {
    return hash_value;
}
```

Hash Function for Class Object

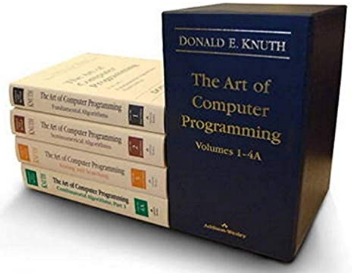
- If we only need the hash value while the object exists in memory, you may use the address:

```
 unsigned int Product::hash() const {
   return reinterpret_cast<unsigned int>( this );
}
```

Check more: https://en.cppreference.com/w/cpp/language/reinterpret_cast

Hash Function for Integer

- ☐ Knuth's Multiplicative Method



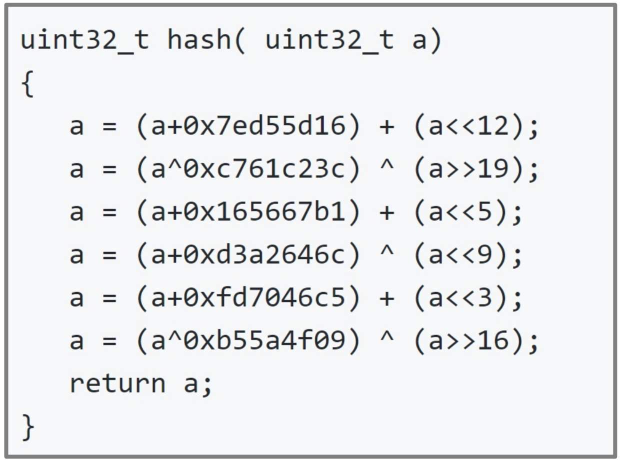
```
hash(i) = i * 2654435761 mod 2^32
```

- 2654435761 is the golden ratio of 2^32
- 2654435761 and 2^32 have no common factors
  - So the multiplication produces a complete mapping of the key to hash result with no overlap
  - Having common factor n would only map to 1/n possible hashes
- Issue: This preserves the divisibility. So for example, if your keys were even, their hashes are always even too.

Reference: Integer hash function, Thomas Wang https://gist.github.com/badboy/6267743

Hash Function for Integer

- Robert Jenkin’s 32-bit integer hash function



```
uint32_t hash( uint32_t a)
{
   a = (a+0x7ed55d16) + (a<<12);
   a = (a^0xc761c23c) ^ (a>>19);
   a = (a+0x165667b1) + (a<<5);
   a = (a+0xd3a2646c) ^ (a<<9);
   a = (a+0xfd7046c5) + (a<<3);
   a = (a^0xb55a4f09) ^ (a>>16);
   return a;
}
```

Reference: Integer hash function, Thomas Wang https://gist.github.com/badboy/6267743

Hash Function for Integer

- It's difficult to tell if your hash function is good or bad
- It depends on the distribution of input keys
- What should be the goodness measure of your hash function?
  - You may need an empirical evaluation?
  - Check "Avalanche effect"
    - One bit change in an input key results in significant changes in an output hash
    - https://en.wikipedia.org/wiki/Avalanche_effect
    - Note: We don't talk about the crypto hash functions here

Hash Function for String

- Two strings are equal if all the characters are equal and in the identical order

- A string is simply an array of bytes:
  - Each byte stores a value from 0 to 255

- Any hash function must be a function of these bytes

Hash Function for String

- We could, for example, just add the characters:

```
unsigned int hash( const string &str ) {
    unsigned int hash_value = 0;

    for ( int k = 0; k < str.length(); ++k ) {
        hash_value += str[k];
    }

    return hash_value;
}
```

Hash Function for String

- Not very good:
  - Words with the same characters hash to the same code:
    - "form" and "from"

Hash Function for String

- Let the individual characters represent the coefficients of a polynomial in *x*:

$$p(x) = c_0 x^{n-1} + c_1 x^{n-2} + \cdots + c_{n-3} x^2 + c_{n-2} x + c_{n-1}$$

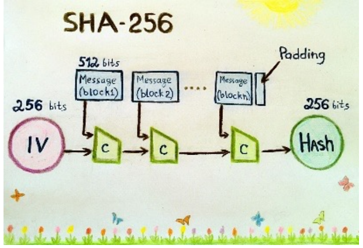
- Use Horner's rule to evaluate this polynomial at a prime number, *e.g.*, *x* = 12347:

```
unsigned int hash( string const &str ) {
    unsigned int hash_value = 0;

    for ( int k = 0; k < str.length(); ++k ) {
        hash_value = 12347*hash_value + str[k];
    }
    return hash_value;
}
```

Hash functions here != Cryptographic Hash

- All the hash functions discussed here are not cryptographic hash functions
  - MD5, SHA-1, SHA-512
  - https://en.wikipedia.org/wiki/Cryptographic_hash_function



https://en.bitcoinwiki.org/wiki/SHA-256

- Cryptographic hash functions have following security properties
  - Pre-image resistance
    - Given h, it is difficult to find m such that h = hash(m)
  - Second pre-image resistance
    - Given m_1, it is difficult to find m_2 such that $\text{hash}(m_1) = \text{hash}(m_2)$
  - Collision resistance
    - Difficult to find any m_1 and m_2 such that $\text{hash}(m_1) = \text{hash}(m_2)$
  - Be careful on the definition of being “difficult”

https://blog.bankofhodlers.com/the-value-of-proof-of-work/

Mapping down to 0, …, M-1

- So far, we computed 32-bit hash values for different input keys
  - Class object
  - Integer
  - String

- Practically, we will require a hash value on the range 0, …, *M*–1:
  - The modulus operator %
  - Review of bitwise operations

Modulus operator

- Easiest method: return the value modulus *M*

```
unsigned int hash_M( unsigned int n, unsigned int M ) {
    return n % M;
}
```

- General modulus operation is expensive

- Modulus operations can be fast if $M = 2^m$
  - Using bitwise/logical operations

Modulus operator with bitwise operations

- 2^m can be represented with a left-shift operation (i.e., 1 << m)

$$2^4 = 10000_2$$

- Modulus operations on 2^m can be represented with a bitwise AND operation

- For example, suppose you want to compute

$$100011100101_2 \,\%\, 10000_2$$

- This is equivalent to zero out all but the last 4 bits using *bitwise AND operation*:

$$1000\;1110\;0101_2 \;\&\; 0000\;0000\;1111_2 \rightarrow 0000\;0000\;0101_2$$

Implementation of Modulus operations

- The implementation using the modulus/remainder operator:

```
unsigned int hash_M( unsigned int n, unsigned int m ) {
    return n & ((1 << m) - 1);
}
```

Summary

- We have seen how a number of objects can be mapped onto a 32-bit integer
- We considered
  - Hash functions for
    - Integer
    - String
    - Class object
  - Map a 32-bit integer onto a smaller range 0, 1, …, $M - 1$

References

[1] Wikipedia, http://en.wikipedia.org/wiki/Hash_function

[2] Cormen, Leiserson, and Rivest, *Introduction to Algorithms*, McGraw Hill, 1990.

[3] Weiss, Data Structures and Algorithm Analysis in C++, 3rd Ed., Addison Wesley.