ECE 250 *Algorithms and Data Structures*

An introduction to hash tables

Douglas Wilhelm Harder, M.Math. LEL
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

ece.uwaterloo.ca
dwharder@alumni.uwaterloo.ca

WATERLOO
ENGINEERING

9.1

Outline

Discuss storing unrelated/unordered data

- IP addresses and domain names

Consider conversions between these two forms

Introduce the idea of hashing:

- Reducing $\mathbf{O}(\ln(n))$ operations to $\mathbf{O}(1)$

Consider some of the weaknesses

9.1.1

Supporting Example

Suppose we have a system which is associated with approximately 150 error conditions where

- Each of which is identified by an 8-bit number from 0 to 255, and
- When an identifier is received, a corresponding error-handling function must be called

We could create an array of 150 function pointers and to then call the appropriate function….

9.1.1.1

Supporting Example

```
#include <iostream>

void a() {
    std::cout
        << "Calling 'void a()'"
        << std::endl;
}

void b() {
    std::cout
        << "Calling 'void b()'"
        << std::endl;
}
```

```
int main() {
    void (*function_array[150])();
    unsigned char error_id[150];

    function_array[0] = a;
    error_id[0] = 3;
    function_array[1] = b;
    error_id[1] = 8;

    function_array[0]();
    function_array[1]();

    return 0;
}
```

Output:

```
% ./a.out
Calling 'void a()'
Calling 'void b()'
```

9.1.1.1

Supporting Example

Unfortunately, this is slow—we would have to do some form of binary search in order to determine which of the 150 slots corresponds to, for example, error-condition identifier `id = 198`

This would require approximately 6 comparisons per error condition

If there was a possibility of dynamically adding new error conditions or removing defunct conditions, this would substantially increase the effort required…

9.1.1.2

Supporting Example

A better solution:

- Create an array of size 256
- Assign those entries corresponding to valid error conditions

```
int main() {
    void (*function_array[256])();
    for ( int i = 0; i < 256; ++i ) {
        function_array[i] = nullptr;
    }

    function_array[3] = a;
    function_array[8] = b;

    function_array[3]();
    function_array[8]();

    return 0;
}
```

Question:

- Is the increased speed worth the allocation of additional memory?

Keys

Our goal:

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

Requirement:

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

In our supporting example, the corresponding function can be called in $\Theta(1)$ time and the array is less than twice the optimal size

9.1.3

Keys

In our example, we:

- Created an array of size 256
- Store each of 150 objects in one of the 256 entries
- The error code indicated which bin the corresponding function pointer was stored

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

9.1.3.1

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

9.1.4

The hash process

We will break the process into three **independent** steps:

- We will try to get each of these down to $\Theta(1)$

Object

Techniques vary...

32-bit integer

Modulus, mid-square, multiplicative, Fibonacci

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

Deal with collisions

Chained hash tables
Open addressing

Linear probing
Quadratic probing
Double hashing

ECE 250 *Algorithms and Data Structures*

Hash functions

Douglas Wilhelm Harder, M.Math. LEL
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

ece.uwaterloo.ca
dwharder@alumni.uwaterloo.ca

WATERLOO
ENGINEERING

Outline

In this talk, we will discuss

- Finding 32-bit hash values using:
  - Predetermined hash values
    - Auto-incremented hash values
    - Address-based hash values
  - Arithmetic hash values
- Example: strings

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, ..., M – 1

9.2.1

The hash process

We will look at the first problem

- Hashing an object into a 32-bit integer
- Subsequent topics will examine the next steps

Object

↓

32-bit integer

↓ Modulus, mid-square, multiplicative, Fibonacci

Map to an index $0, \ldots, M - 1$

↓

Deal with collisions

Chained hash tables
Open addressing

Linear probing
Quadratic probing
Double hashing

9.2.2

Properties

Necessary properties of such a hash function h are:

1a. Should be fast: ideally $\Theta(1)$

1b. The hash value must be *deterministic*

- It must always return the same 32-bit integer each time

1c. Equal objects hash to equal values

- $x = y \quad \Rightarrow \quad h(x) = h(y)$

1d. If two objects are randomly chosen, there should be only a one-in-2^{32} chance that they have the same hash value

9.2.3

Types of hash functions

We will look at two classes of hash functions

- Predetermined hash functions (explicit)
- Arithmetic hash functions (implicit)

9.2.4

Predetermined hash functions

The easiest solution is to give each object a unique number

```
class Class_name {
    private:
        unsigned int hash_value;  // int:            -2^31, ..., 2^31 – 1
                                  // unsigned int:    0, ..., 2^32 – 1
    public:
        Class_name();
        unsigned int hash() const;
};
```

```
Class_name::Class_name() {
    hash_value = ???;
}
```

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

9.2.4 Predetermined hash functions

For example, an auto-incremented static member variable

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

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

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

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

9.2.4 Predetermined hash functions

Examples: All UW co-op student have two hash values:
- UW Student ID Number
- Social Insurance Number

Any 9-digit-decimal integer yields a 32-bit integer

$$\lg(10^9) = 29.897$$

9.2.4

Predetermined hash functions

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

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

This fails if an object may be stored in secondary memory

– It will have a different address the next time it is loaded

9.2.4.1

Predetermined hash functions

Predetermined hash values give each object a unique hash value

This is not always appropriate:

- Objects which are conceptually equal:

```
Rational x(1, 2);
Rational y(3, 6);
```

- Strings with the same characters:

```
string str1 = "Hello world!";
string str2 = "Hello world!";
```

These hash values must depend on the member variables

- Usually this uses arithmetic functions

9.2.5

Arithmetic Hash Values

An arithmetic hash value is a deterministic function that is calculated from the relevant member variables of an object

We will look at arithmetic hash functions for:

- Rational numbers, and
- Strings

9.2.5.1

Rational number class

What if we just add the numerator and denominator?

```
class Rational {
    private:
        int numer, denom;
    public:
        Rational( int, int );
};

unsigned int Rational::hash() const {
    return static_cast<unsigned int>( numer ) +
        static_cast<unsigned int>( denom );
}
```

9.2.5.1

Rational number class

We could improve on this: multiply the denominator by a large prime:

```
class Rational {
    private:
        int numer, denom;
    public:
        Rational( int, int );
};

unsigned int Rational::hash() const {
    return static_cast<unsigned int>( numer ) +
        429496751*static_cast<unsigned int>( denom );
}
```

9.2.5.1

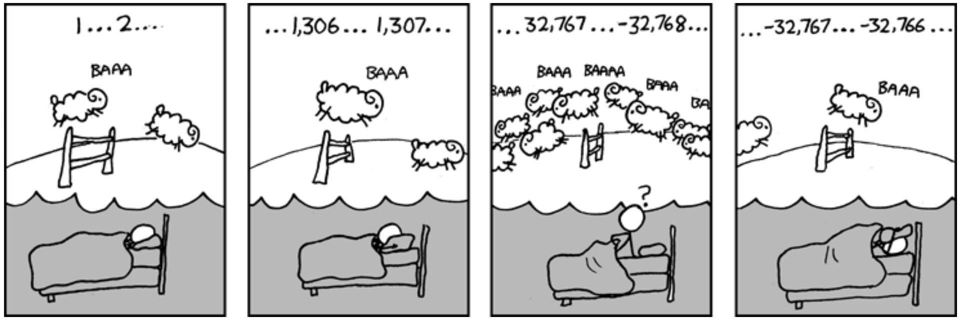
Rational number class

For example, the output of

```
int main() {
    cout << Rational(  0,   1 ).hash() << endl;
    cout << Rational(  1,   2 ).hash() << endl;
    cout << Rational(  2,   3 ).hash() << endl;
    cout << Rational( 99, 100 ).hash() << endl;

    return 0;
}
```

is

```
429496751
858993503
1288490255
2239
```



http://xkcd.com/571/

Recall that arithmetic operations wrap on overflow

9.2.5.1

Rational number class

This hash function does not generate unique values

- The following pairs have the same hash values:

0/1	1327433019/800977868
1/2	534326814/1480277007
2/3	820039962/1486995867

- Finding rational numbers with matching hash values is very difficult:
- Finding these required the generation of 1 500 000 000 random rational numbers
- It is fast: $\Theta(1)$
- It does produce an even distribution

9.2.5.1

Rational number class

Problem:

- The rational numbers 1/2 and 2/4 have different values
- The output of

```
int main() {
    cout << Rational( 1, 2 ).hash();
    cout << Rational( 2, 4 ).hash();
    return 0;
}
```

is

```
858993503
1717987006
```

9.2.5.1

Rational number class

Solution: divide through by the greatest common divisor

```
Rational::Rational( int a, int b ):numer(a), denom(b) {
    int divisor = gcd( numer, denom );
    numer /= divisor;
    denom /= divisor;
}
```

```
int gcd( int a, int b) {
    while( true ) {
        if ( a == 0 ) {
            return (b >= 0) ? b : -b;
        }

        b %= a;

        if ( b == 0 ) {
            return (a >= 0) ? a : -a;
        }
        a %= b;
    }
}
```

9.2.5.1
Rational number class

Problem:

- The rational numbers $\frac{1}{2}$ and $\frac{-1}{-2}$ have different values
- The output of

```
int main() {
    cout << Rational(  1,  2 ).hash();
    cout << Rational( -1, -2 ).hash();
    return 0;
}
```

is

```
858993503
3435973793
```

9.2.5.1

Rational number class

Solution: define a normal form

- Require that the denominator is positive

```
Rational::Rational( int a, int b ):numer(a), denom(b) {
    int divisor = gcd( numer, denom );
    divisor = (denom >= 0) ? divisor : -divisor;
    numer /= divisor;
    denom /= divisor;
}
```

String class

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

9.2.5.3.1

String class

We could, for example, just add the characters:

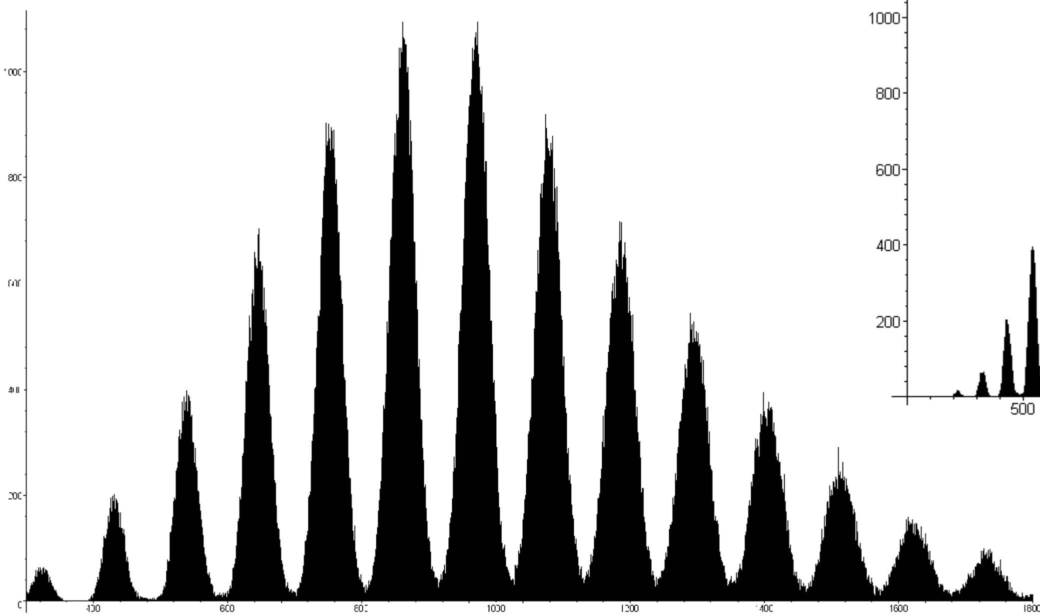
```
unsigned int hash( const string &str ) {
    unsigned int hash_vaalue = 0;

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

    return hash_value;
}
```

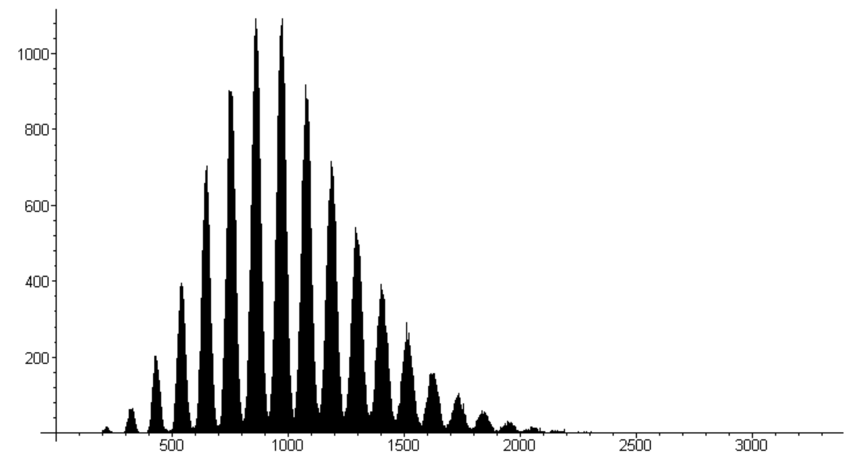
9.2.5.3.1

String class

Not very good:

- Slow run time: $\Theta(n)$
- Words with the same characters hash to the same code:
  - `"form"` and `"from"`
- A poor distribution, *e.g.*, all words in Moby™ Words II by Grady Ward (`single.txt`) Project Gutenberg):





9.2.5.3.2

String class

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;
}
```

9.2.5.3.3

Arithmetic hash functions

In general, any member variables that are used to uniquely define an object may be used as coefficients in such a polynomial

- The salary hopefully changes over time…

```
class Person {
    string surname;
    string *given_names;
    unsigned char num_given_names;
    unsigned short birth_year;
    unsigned char birth_month;
    unsigned char birth_day;
    unsigned int salary;
    // ...
};
```

Summary

We have seen how a number of objects can be mapped onto a 32-bit integer

We considered

- Predetermined hash functions
  - Auto-incremented variables
  - Addresses
- Hash functions calculated using arithmetic

Next: map a 32-bit integer onto a smaller range $0, 1, ..., M - 1$