

# CS101 Algorithms and Data Structures

Hash Table  
Textbook Ch 11



# Outline

- Introduction
- Hash function
- Mapping down to  $0, \dots, M - 1$
- Dealing with collisions
  - Chained hash tables
  - Open addressing

# Supporting Example

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

- Each of which is identified by an 16-bit number from 0 to 65535, 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....

# Supporting Example

Given an error-condition identifier, e.g., `id = 198`, how shall we determine which of the 150 slots corresponds to it?

- Binary search!

## Problems

- This is slow: it would require approximately 7 comparisons per error condition
- Slow to dynamically add new error conditions or remove defunct conditions

# IP Addresses

Examples:

Suppose we want to associate IP addresses and any corresponding domain names

Recall that a 32-bit IP address are often written as four byte values from 0 to 255

- Consider 10000001 01100001 00001010 10110011<sub>2</sub>
- This can be written as 129.97.10.179
- We use domain names because IP addresses are not human readable

# IP Addresses

Given an IP address, sometimes we wanted to *quickly* find any associated domain name.

We could create an array of size  $2^{32} = 4,294,967,296$  of strings!

```
string domain_name[4294967296];
```

For example, the IP address of shanghaitech.edu.cn is 10.15.42.202

– As  $202 + 42 \times 2^8 + 15 \times 2^{16} + 10 \times 2^{24} = 168766154$ , it follows that

```
domain_name[168766154] = "shanghaitech.edu.cn";
```

# IP Addresses

Given an IP address, sometimes we wanted to *quickly* find any associated domain name.

We could create an array of size  $2^{32} = 4,294,967,296$  of strings!

```
string domain_name[4294967296];
```

As of 2015, the number of domain names is 299 million.

So most part of the array is empty!

# Goal

Our goal:

- Store data so that all operations are  $\Theta(1)$  time
- The memory requirement should be  $\Theta(n)$



# Simpler problem

Let's try a simpler problem

- How do I store your examination grades so that I can access your grades in  $\Theta(1)$  time?

Recall that each student is issued an 8-digit number

- How do I store your examination grades so that I can access your grades in  $\Theta(1)$  time?
- Create an array of size  $10^8 \approx 1.5 \times 2^{26}$  ?

# Simpler problem

I could create an array of size 1000

- How could you convert an 8-digit number into a 3-digit number?
- Idea: the last three digits, which seem random

Therefore, I could store the examination grade of student “10105456” by:

```
grade[456] = 86;
```

# Simpler problem

Question:

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

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

# Simpler problem

Consequently, I have a function that maps a student onto a 3-digit number

- I can store the examination grade in that location
- Storing it, accessing it, and erasing it is  $\Theta(1)$
- Problem: two or more students may map to the same number:
  - Student A has ID 20173456 and scored 85
  - Student B has ID 20234456 and scored 87

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

# 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

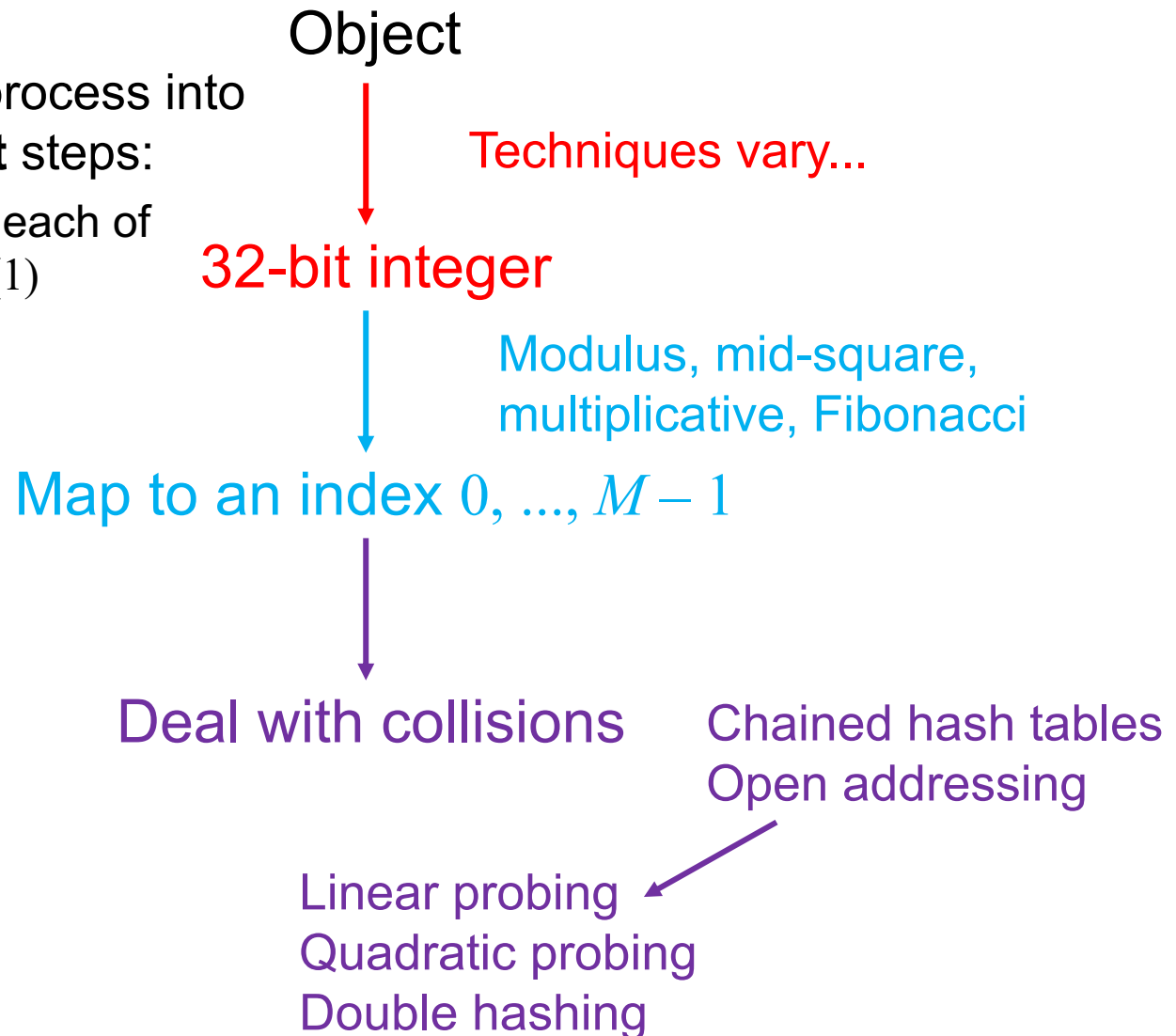
- Such an event is termed a *collision*

Hash tables use a hash function together with a mechanism for dealing with collisions

# The hash process

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

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



# Outline

- Introduction
- Hash function
- Mapping down to  $0, \dots, M - 1$
- Dealing with collisions
  - Chained hash tables
  - Open addressing

# 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**



# 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

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

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

# Types of hash functions

We will look at two classes of hash functions

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

# 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;  
}
```

# Predetermined hash functions

- Problem with predetermined hash functions?
  - Strings with the same characters:  
`string str1 = "Hello world!";`  
`string str2 = "Hello world!";`
  - Objects which are conceptually equal:  
`Rational x(1, 2);`  
`Rational y(3, 6);`
- The previous two methods would give them different hash values.
- But, a hash function should “hash equal objects to equal values”
- These hash values must depend on the member variables
  - Usually this uses arithmetic functions

# 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

# String class

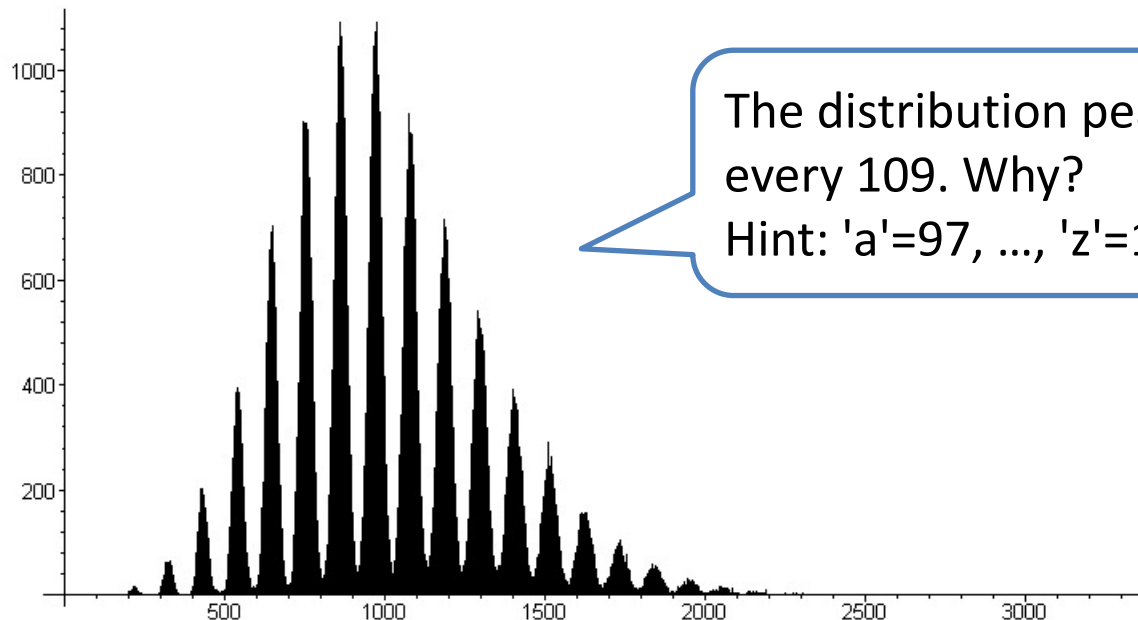
We could, for example, just add the characters:

```
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;  
}
```

# 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:



The distribution peaks about every 109. Why?

Hint: 'a'=97, ..., 'z'=122

# 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;  
}
```



# String class

Problem, Horner's rule runs in  $\Theta(n)$

```
"A Elbereth Gilthoniel,\n  Silivren penna miriel\n  O menal aglar elenath!\n  Na-chaered palan-diriel\n  O galadhremmin ennorath,\n  Fanuilos, le linnathon\n  nef aear, si nef aearon!"
```

Suggestions?

# String class

Use characters in locations  $2^k - 1$  for  $k = 0, 1, 2, \dots$ :

```
"A_Elbereth Giltthoniel,\nSilivren_penna miriel\nO menal aglar elenath!\nNa-chaered palan-diriel\nO galadhremmin ennorath,\nFanuilos, le linnathon\nnef aear, si nef aearon!"
```

J.R.R. Tolkien

# String class

The run time is now  $\Theta(\ln(n))$  :

```
unsigned int hash( const string &str ) {  
    unsigned int hash_value = 0;  
  
    for ( int k = 1; k <= str.length(); k *= 2 ) {  
        hash_value = 12347*hash_value + str[k - 1];  
    }  
  
    return hash_value;  
}
```

# Arithmetic hash functions

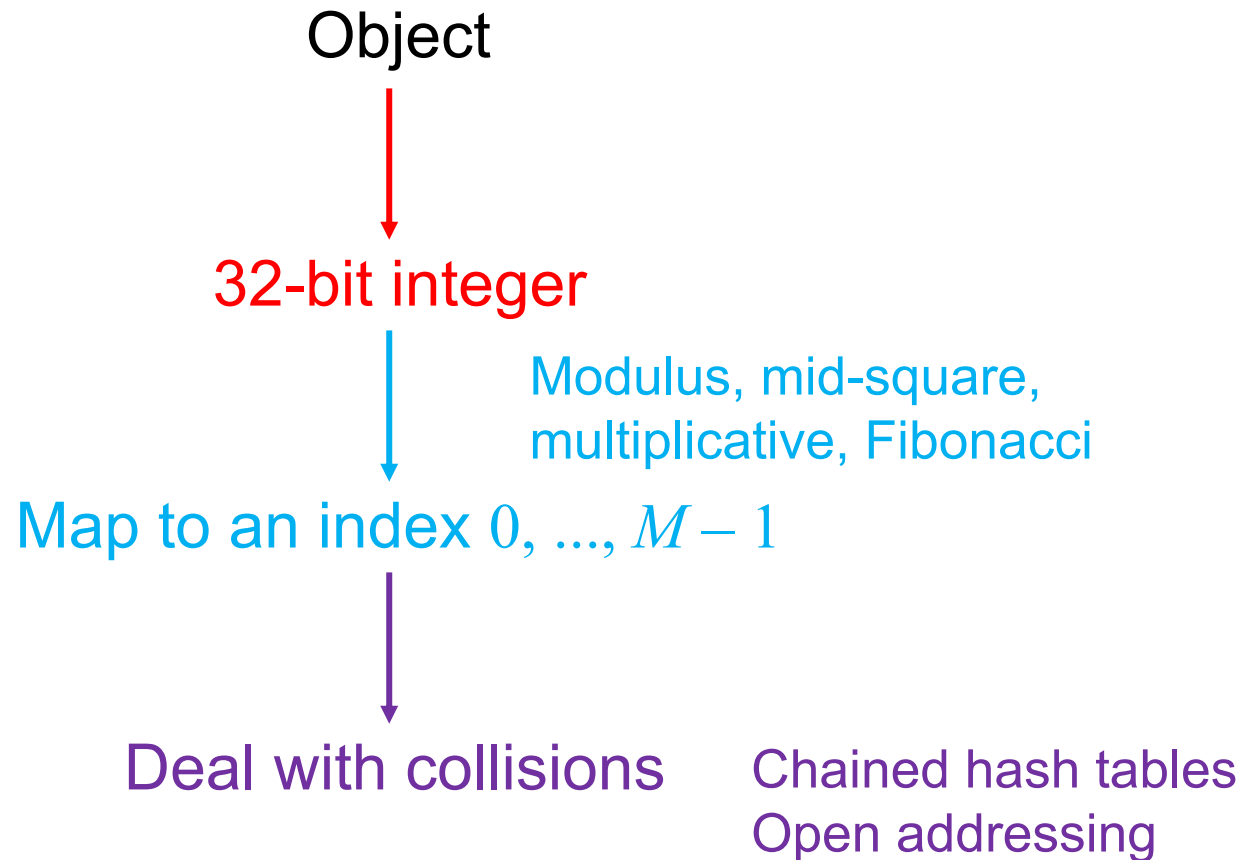
In general, any member variables that are used to uniquely define an object may be used as coefficients in such a polynomial

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

# Outline

- Introduction
- Hash function
- Mapping down to  $0, \dots, M - 1$
- Dealing with collisions
  - Chained hash tables
  - Open addressing

# The hash process



# Properties

Necessary properties of this mapping function  $h_M$  are:

2a. Must be fast:  $\Theta(1)$

2b. The hash value must be *deterministic*

- Given  $n$  and  $M$ ,  $h_M(n)$  must always return the same value

2c. If two objects are randomly chosen, there should be only a one-in- $M$  chance that they have the same value from 0 to  $M - 1$

# Modulus operator

Easiest method: return the value modulus  $M$

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

Unfortunately, calculating the modulus (or remainder) is expensive

- If  $M = 2^m$ , we can simplify the calculation by bitwise operations
  - left and right shift and bit-wise and



# The bitwise operators: & << >>

Suppose I want to calculate

$$7985325 \% 100$$

The modulo is a power of ten:  $100 = 10^2$

- In this case, take the last **two** decimal digits: 25

Similarly,  $7985325 \% 10^3 = 325$

- We set the appropriate digits to 0:

**0000**25 and **000**325

# The bitwise operators: & << >>

The same works in base 2:

$$100011100101_2 \% 10000_2$$

The modulo is a power of 2:  $10000_2 = 2^4$

- In this case, take the last **four** bits: 0101

Similarly,  $100011100101_2 \% 1000000_2 == 100101$ ,

- We set the appropriate digits to 0:

$$\text{00000000}0101 \text{ and } \text{000000}100101$$

# The bitwise operators: & << >>

To zero all but the last  $n$  bits, select the last  $n$  bits using *bitwise and*:

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

$1000\ 1110\ 0101_2 \ \&\ 0000\ 0011\ 1111_2 \rightarrow 0000\ 0010\ 0101_2$

# The bitwise operators: & << >>

Similarly, multiplying or dividing by powers of 10 is easy:

$$7985325 * 100$$

The multiplier is a power of ten:  $100 = 10^2$

- In this case, add **two** zeros: 798532500

Similarly,  $7985325 / 10^3 = 7985$

- Just add the appropriate number of zeros or remove the appropriate number of digits

# The bitwise operators: & << >>

The same works in base 2:

$$100011100101_2 * 10000_2$$

The modulo is a power of 2:  $10000_2 = 2^4$

– In this case, add **four** zeros:  $1000111001010000$

Similarly,  $100011100101_2 / 1000000_2 == 100011$

# The bitwise operators: & << >>

This can be done mechanically by shifting the bits appropriately:

$$1000\ 1110\ 0101_2 \ll 4 == 1000\ 1110\ 0101\ 0000_2$$

$$1000\ 1110\ 0101_2 \gg 6 == 10\ 0011_2$$

Powers of 2 are now easy to calculate:

$$1_2 \ll 4 == 1\ 0000_2 \quad // \quad 2^4 = 16$$

$$1_2 \ll 6 == 100\ 0000_2 \quad // \quad 2^6 = 64$$

# Modulo a power of two

The implementation using the modulus/remainder operator:

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

# Modulo a power of two

Problem:

- Suppose that the hash function  $h$  is always even
- An even number modulo a power of two is still even

Example: memory allocations are multiples of word size

- On a 64-bit computer, addresses returned by new will be multiples of 8
- The probability that  $h_M(h(x)) = h_M(h(y))$  is one in  $M/8$ 
  - This is not one in  $M$



# The multiplicative method

We need to obfuscate the bits

- The most common method to obfuscate bits is multiplication
- Consider how one bit can affect an entire range of numbers in the result:

$$\begin{array}{r} 10100111 \\ \times 11010011 \\ \hline 10100111 \\ 10100111 \\ 10100111 \\ 10100111 \\ 10100111 \\ + 10100111 \\ \hline 1000101110100101 \end{array}$$

The *avalanche* effect:  
changing one bits has the  
potential of affecting all bits  
in the result:

$$\begin{aligned} 10100011 \times 11010011 \\ = 1000011001011001 \end{aligned}$$

# The multiplicative method

Multiplying by a fixed constant is a reasonable method

- Take the middle  $m$  bits of  $Cn$ :

```
unsigned int const C = 581869333;  // some number
```

```
unsigned int hash_M( unsigned int n, unsigned int m ) {  
    unsigned int shift = (32 - m)/2;  
    return ((C*n) >> shift) & ((1 << m) - 1);  
}
```

# The multiplicative method

Suppose that the value  $m = 10$  ( $M = 1024$ ) and  $n = 42$

```
const unsigned int C = 581869333; // some number
```

```
unsigned int hash_M( unsigned int n, unsigned int m ) {  
    unsigned int shift = (32 - m)/2;  
    return ((C*n) >> shift) & ((1 << m) - 1);  
}
```

# The multiplicative method

$m = 10$

$n = 42$

First calculate the shift

```
const unsigned int C = 581869333; // some number
```

```
unsigned int hash_M( unsigned int n, unsigned int m ) {  
    unsigned int shift = (32 - m)/2;  
    return ((C*n) >> shift) & ((1 << m) - 1);  
}
```

shift = 11

# The multiplicative method

$m = 10$

$n = 42$

Calculate  $Cn$

```
const unsigned int C = 581869333; // some number
```

```
unsigned int hash_M( unsigned int n, unsigned int m ) {  
    unsigned int shift = (32 - m)/2;  
    return ((C*n) >> shift) & ((1 << m) - 1);  
}
```

**shift = 11**

1	0	1	1	0	0	0	0	1	0	1	0	0	1	1	0	0	0	0	1	1	0	0	1	0	1	1	1	0	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# The multiplicative method

$m = 10$

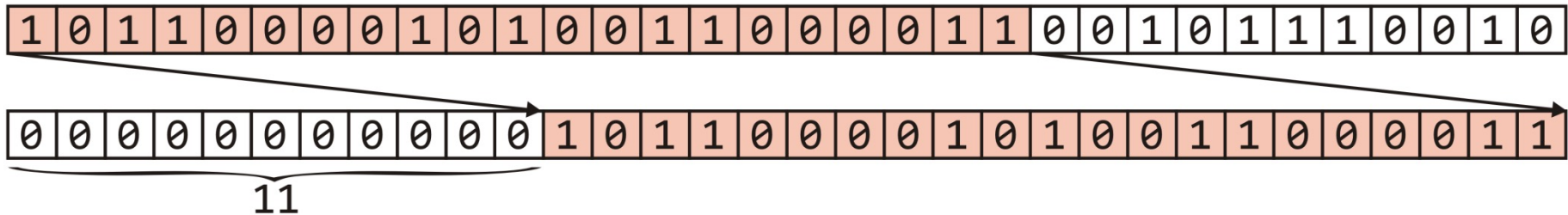
$n = 42$

Right shift this value 11 bits—equivalent to dividing by  $2^{11}$

```
const unsigned int C = 581869333; // some number
```

```
unsigned int hash_M( unsigned int n, unsigned int m ) {  
    unsigned int shift = (32 - m)/2;  
    return ((C*n) >> shift) & ((1 << m) - 1);  
}
```

**shift = 11**



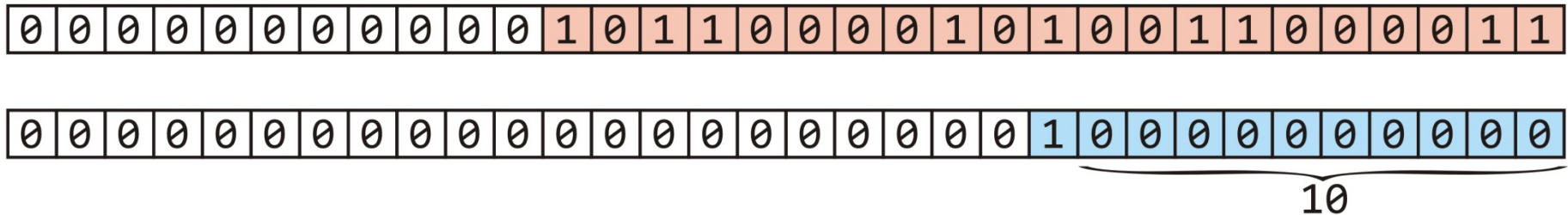
# The multiplicative method

$$m = 10$$
$$n = 42$$

Left shift 1  $m = 10$  bits yielding  $2^{10}$

```
const unsigned int C = 581869333; // some number
```

```
unsigned int hash_M( unsigned int n, unsigned int m ) {
    unsigned int shift = (32 - m)/2;
    return ((C*n) >> shift) & ((1 << m) - 1);
}
```



# The multiplicative method

Subtracting 1 yields  $m = 10$  ones

$$m = 10$$
$$n = 42$$

```
const unsigned int C = 581869333; // some number
```

```
unsigned int hash_M( unsigned int n, unsigned int m ) {
    unsigned int shift = (32 - m)/2;
    return ((C*n) >> shift) & ((1 << m) - 1);
}
```

0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0	0	0	1	0	1	0	0	1	1	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

[illegible]



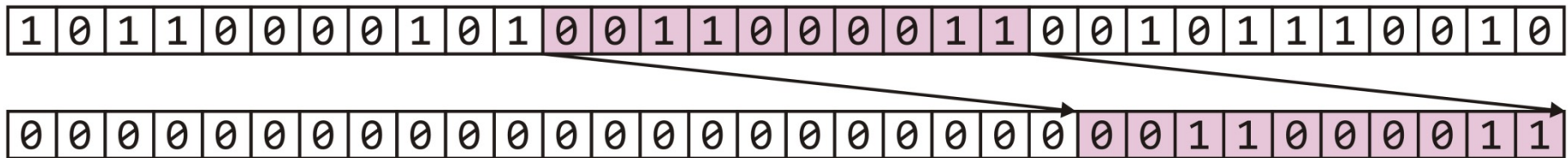


# The multiplicative method

We have extracted the middle  $m = 10$  bits—a number in  $0, \dots, 1023$

```
const unsigned int C = 581869333; // some number
```

```
unsigned int hash_M( unsigned int n, unsigned int m ) {  
    unsigned int shift = (32 - m)/2;  
    return ((C*n) >> shift) & ((1 << m) - 1);  
}
```

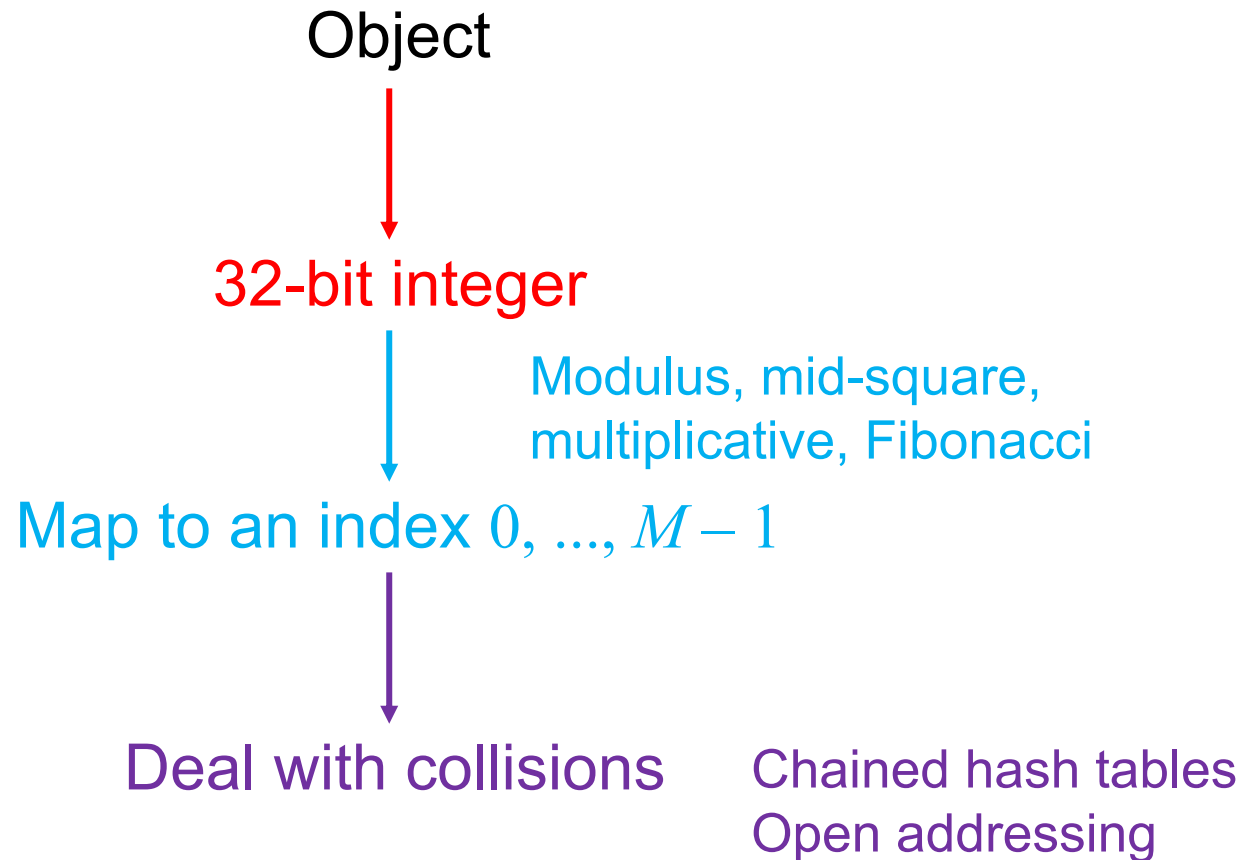


$$h_M(42) = 195$$

# Outline

- Introduction
- Hash function
- Mapping down to  $0, \dots, M - 1$
- Dealing with collisions
  - Chained hash tables
  - Open addressing

# The hash process



# Chained hash table

Associating each bin with a linked list.

For any object assigned to the bin by the hash function, finding, inserting, and erasing the object is done on the linked list.

# Example

As an example, let's store hostnames and allow a fast look-up of the corresponding IP address

- We will choose the bin based on the host name
- Associated with the name will be the IP address
- *E.g.*, ("optimal", 129.97.94.57)

# Example

Suppose the hash value of a string is the last 3 bits of the first character in the host name

- The hash of "optimal" is based on "o"

a	01100001	n	01101110
b	01100010	o	01101111
c	01100011	p	01110000
d	01100100	q	01110001
e	01100101	r	01110010
f	01100110	s	01110011
g	01100111	t	01110100
h	01101000	u	01110101
i	01101001	v	01110110
j	01101010	w	01110111
k	01101011	x	01111000
l	01101100	y	01111001
m	01101101	z	01111010

# Example

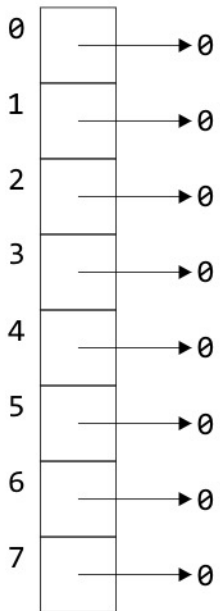
Our hash function is

```
unsigned int hash( string const &str ) {  
    // the empty string "" is hashed to 0  
    if str.length() == 0 ) {  
        return 0;  
    }  
  
    return str[0] & 7;  
}
```



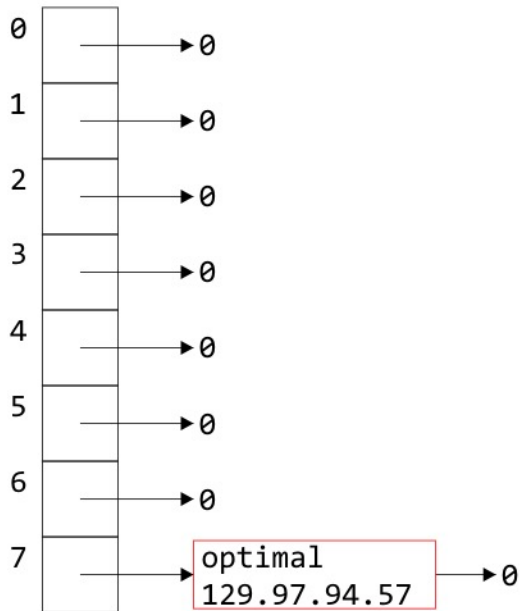
# Example

Starting with an array of 8 empty linked lists



# Example

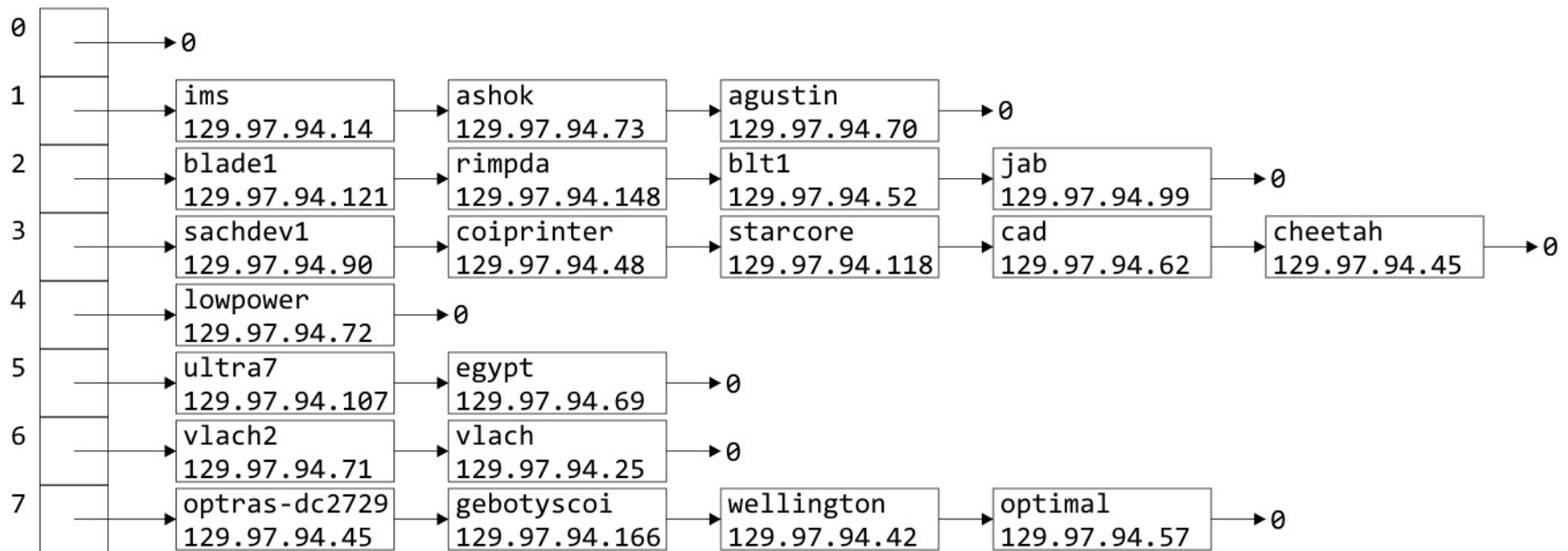
The pair ("optimal", 129.97.94.57) is entered into bin  
01101**111** = 7



# Example

Indeed, after 21 insertions, the linked lists are becoming rather long

- We were looking for  $\Theta(1)$  access time, but accessing something in a linked list with  $k$  objects is  $\mathcal{O}(k)$



# Load Factor

To describe the length of the linked lists, we define the *load factor* of the hash table:

$$\lambda = \frac{n}{M}$$

This is the average number of objects per bin

- This assumes an even distribution

Right now, the load factor is  $\lambda = 21/8 = 2.625$

- The average bin has 2.625 objects

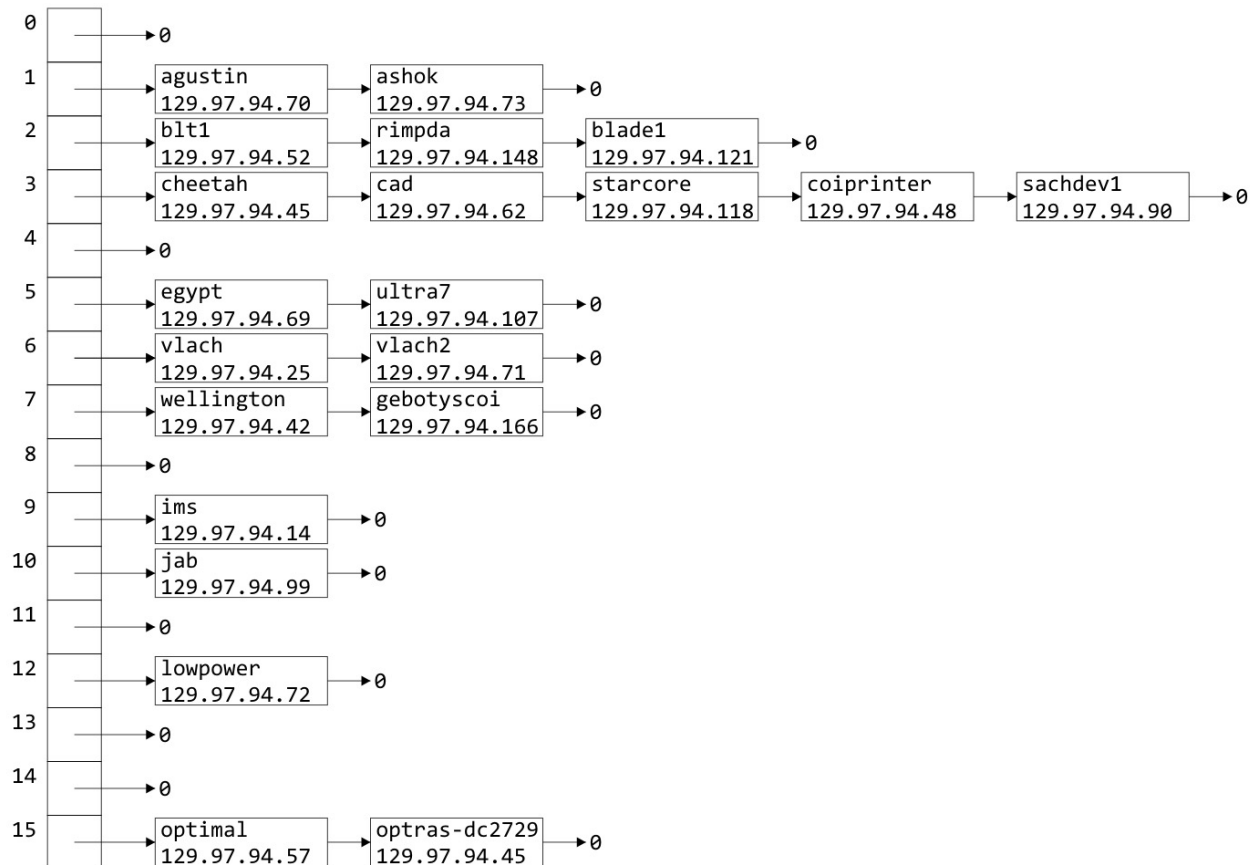
# Load Factor

If the load factor becomes too large, access times will start to increase:  $O(\lambda)$

The most obvious solution is to double the size of the hash table and re-insert every object (*rehashing*)

# Doubling Size

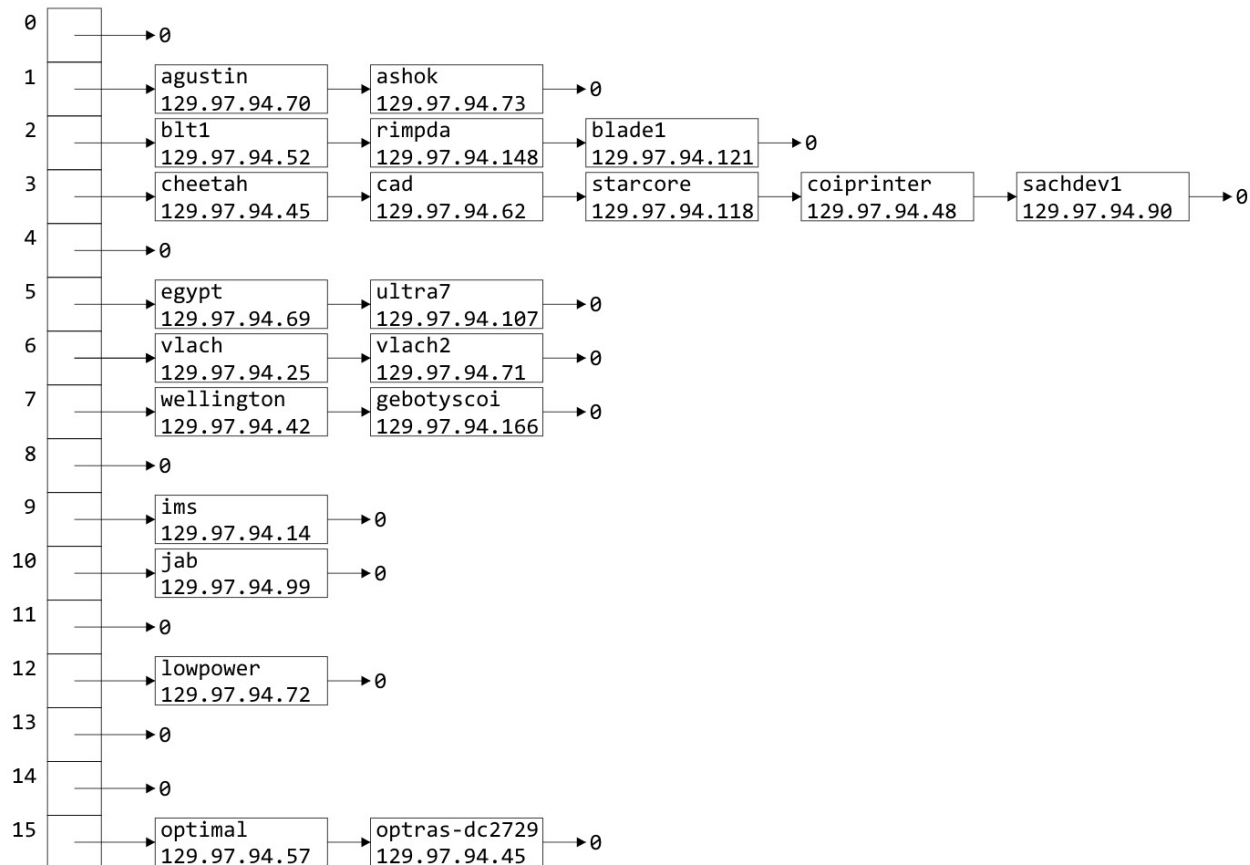
In our example, suppose we take the last four bits as the hash function after doubling the hash table size



# Doubling Size

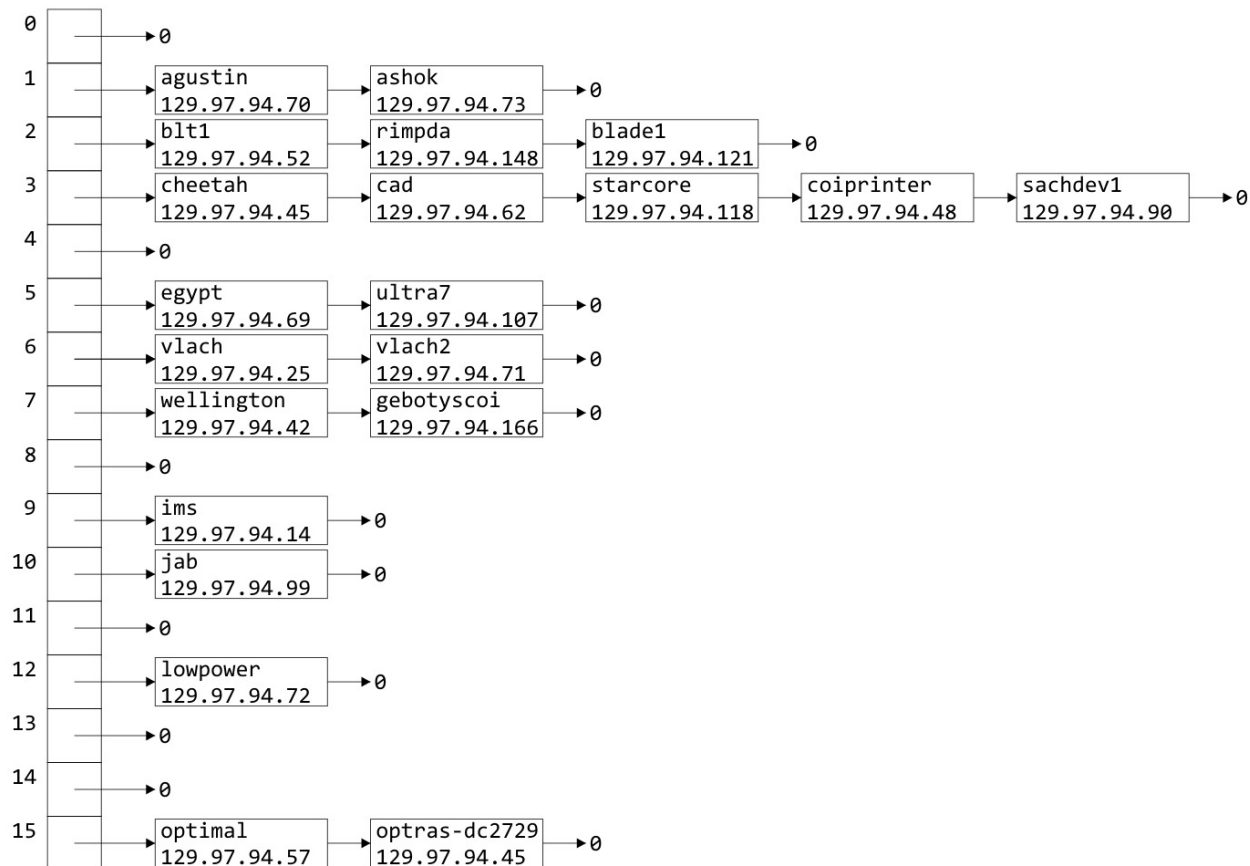
The load factor is now  $\lambda = 1.3125$

- Unfortunately, the distribution hasn't improved much



# Doubling Size

There is significant *clustering* in bins 2 and 3 due to the choice of host names





# Choosing a Good Hash Function

We choose a very poor hash function:

- We looked at the first letter of the host name

Unfortunately, all these are also actual host names:

ultra7 ultra8 ultra9 ultra10 ultra11

ultra12 ultra13 ultra14 ultra15 ultra16 ultra17

blade1 blade2 blade3 blade4 blade5

This will cause clustering in bins 2 and 5

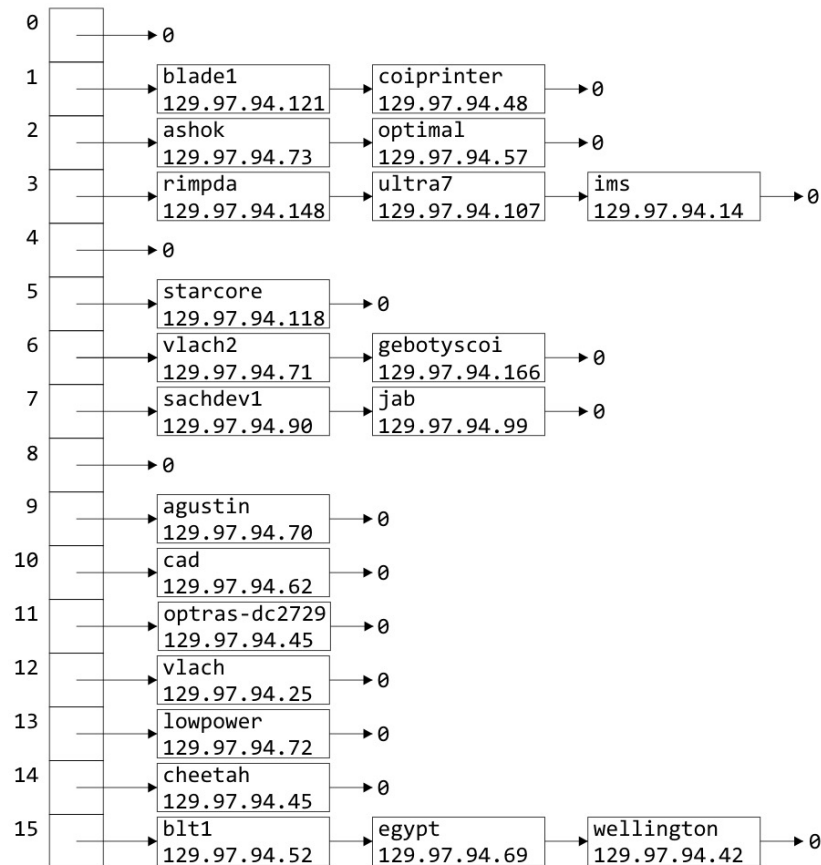
# Choosing a Good Hash Function

Let's go back to the hash function defined previously:

```
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;  
}
```

# Choosing a Good Hash Function

This hash function yields a much nicer distribution:



# Problems with Linked Lists

One significant issue with chained hash tables using linked lists

- It requires extra memory
- It uses dynamic memory allocation

Another issue is the  $O(\lambda)$  time complexity

For faster access, **we could replace each linked list with an AVL tree** (assuming we can order the objects)

- The access time drops to  $O(\ln(\lambda))$
- The memory requirements are increased by  $\Theta(n)$ , as each node will require two pointers

# Outline

- Introduction
- Hash function
- Mapping down to  $0, \dots, M - 1$
- Dealing with collisions
  - Chained hash tables
  - Open addressing

# Background

Chained hash tables require special memory allocation

- Can we create a hash table without significant memory allocation?

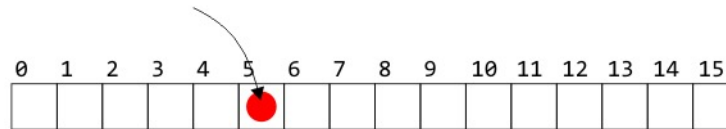
We will deal with collisions by storing collisions elsewhere

- We will define an implicit rule which tells us where to look next

# Open Addressing

Suppose an object hashes to bin 5

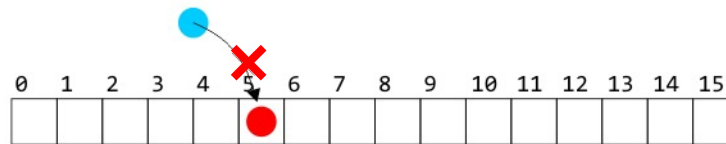
- If bin 5 is empty, we can copy the object into that entry



# Open Addressing

Suppose, however, another object hashes to bin 5

- Without a linked list, we cannot store the object in that bin

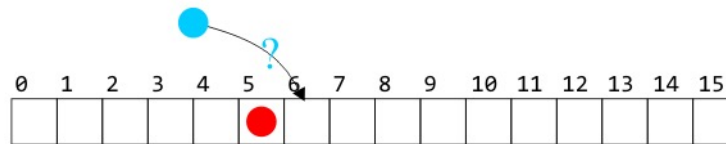




# Open Addressing

We need a rule to tells us where to look next

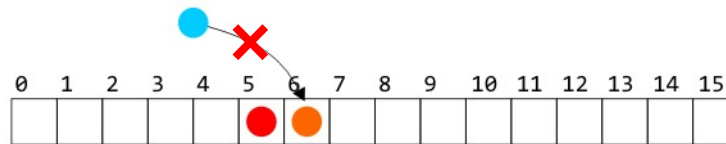
- For example, look in the next bin to see if it is occupied



# Open Addressing

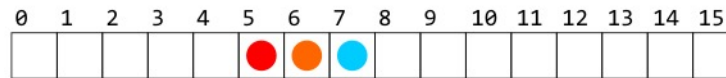
The rule must be:

- simple to follow—*i.e.*, fast
- general enough to deal with the fact that the next cell could also be occupied: e.g., continue searching until the first empty bin is found



# Open Addressing

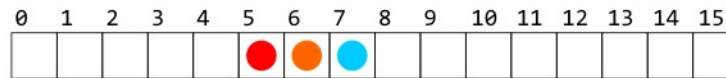
Of course, whatever rule we use in placing an object must also be used when searching for or removing objects



# Open Addressing

Recall, however, that our goal is  $\Theta(1)$  access times

- We cannot, on average, be forced to access too many bins



# Open Addressing

There are numerous strategies for defining the order in which the bins should be searched:

- Linear probing
- Quadratic probing
- Double hashing

There are many alternate strategies, as well:

- Last come, first served
  - Always place the object into the bin moving what may be there already
- Cuckoo hashing

# Outline

- Introduction
- Hash function
- Mapping down to  $0, \dots, M - 1$
- Dealing with collisions
  - Chained hash tables
  - Open addressing
    - Linear probing
    - Quadratic probing

# Linear Probing

The easiest method to probe the bins of the hash table is to search forward linearly

Assume we are inserting into bin  $k$ :

- If bin  $k$  is empty, we occupy it
- Otherwise, check bin  $k + 1$ ,  $k + 2$ , and so on, until an empty bin is found
  - If we reach the end of the array, we start at the front (bin 0)

# Linear Probing

Consider a hash table with  $M = 16$  bins

Given a 3-digit hexadecimal number:

- The least-significant digit is the primary hash function (bin)
- Example: for  $72A_{16}$ , the initial bin is **A**



# Insertion

Insert these numbers into this initially empty hash table:

19A, 207, 3AD, 488, 5BA, 680, 74C, 826, 946, ACD, B32, C8B, D59, E9C

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

## Example

Start with the first four values:

19A, 207, 3AD, 488

[illegible]

# Example

Start with the first four values:

19**A**, 20**7**, 3A**D**, 48**8**

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							207	488		19A			3AD		

# Example

Next we must insert 5BA

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							207	488		19A			3AD		

# Example

Next we must insert 5B**A**

- Bin **A** is occupied
- We search forward for the next empty bin

0	1	2	3	4	5	6	7	8	9	<b>A</b>	B	C	D	E	F
							207	488		19A	<b>5BA</b>		3AD		

# Example

Next we are adding 680, 74C, 826

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							207	488		19A	5BA		3AD		

# Example

Next we are adding 680, 74C, 826

- All the bins are empty—simply insert them

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680						826	207	488		19A	5BA	74C	3AD		

# Example

Having completed these insertions:

- The load factor is  $\lambda = 14/16 = 0.875$
- The average number of probes is  $38/14 \approx 2.71$

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E9C			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B



# Resizing the array

To double the capacity of the array, each value must be rehashed

- We use the **least-significant five bits** for the initial bin
- 680, B32, ACD, 5BA, 826, 207, 488, D59 may be immediately placed

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
680						826	207	488					ACD					B32							D59	5BA					

# Resizing the array

To double the capacity of the array, each value must be rehashed

- Both E9C and C8B fit without a collision
- The load factor is  $\lambda = 14/32 = 0.4375$
- The average number of probes is  $18/14 \approx 1.29$

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
680						826	207	488	946		C8B	74C	ACD	3AD			946	B32							D59	5BA	19A	E9C			

# Searching

Testing for membership is similar to insertions:

Start at the appropriate bin, and searching forward until

1. The item is found,
2. An empty bin is found, or
3. We have traversed the entire array

The third case will only occur if the hash table is full (load factor of 1)

# Searching

Searching for C8B

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Searching

Searching for C8**B**

- Examine bins B, C, D, E, F
- The value is found in F

0	1	2	3	4	5	6	7	8	9	A	<b>B</b>	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	<b>C8B</b>

# Erasing

Can we simply remove elements from the hash table?

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Erasing

We cannot simply remove elements from the hash table

- For example, consider erasing 3AD

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Erasing

We cannot simply remove elements from the hash table

- For example, consider erasing 3AD
- If we just erase it, it is now an empty bin
  - By our algorithm, we cannot find ACD, C8B and D59

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C		ACD	C8B



# Erasing

Instead, we must attempt to fill the empty bin

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C		ACD	C8B

# Erasing

Instead, we must attempt to fill the empty bin

- We can move ACD into the location
- Are we done?

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	<del>ACB</del>	ACD	C8B

# Erasing

Now we have another bin to fill

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	ACD		C8B

# Erasing

Now we have another bin to fill

- We can move C8B into the location

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	ACD	<del>C8B</del>	C8B

# Erasing

Now we must attempt to fill the bin at F

- We cannot move 680

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	ACD	C8B	

# Erasing

Now we must attempt to fill the bin at F

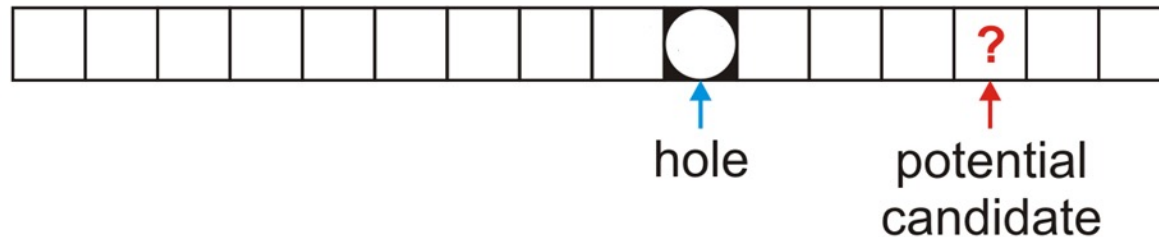
- We cannot move 680
- We can, however, move D59

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	ACD	C8B	<b>D59</b>

# Erasing

In general, assume:

- The currently removed object has created a hole at index **hole**
- The object we are checking is located at the position **index** and has a hash value of hash

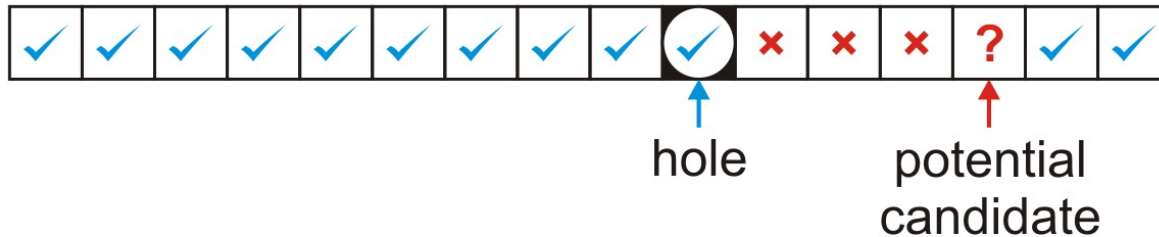


- Remember: if we are checking the object **?** at location **index**, this means that all entries between **hole** and **index** are both occupied and could not have been copied into the hole

# Erasing

The first possibility is that  $\text{hole} < \text{index}$

- In this case, we move the object at `index` only if its hash value is either
  - equal to or less than the `hole` **or**
  - greater than the `index` of the potential candidate





# Erasing

The other possibility is we wrapped around the end of the array, that is,  $\text{hole} > \text{index}$

- In this case, we move the object at  $\text{index}$  only if its hash value is both
  - greater than the index of the potential candidate **and**
  - less than or equal to the hole



In either case, if the move is successful, the **?** now becomes the new hole to be filled

# Alternative Method: Lazy Erasing

- Consider erasing 3AD

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	3AD	ACD	C8B

# Alternative Method: Lazy Erasing

- Consider erasing 3AD
  - Mark the bin as ERASED
  - Searching: regard it as occupied
  - Insertion: regard it as unoccupied
    - What if we want to insert ACD?
    - Search before insertion

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
680	D59	B32	E93			826	207	488	946	19A	5BA	74C	<del>3AD</del>	ACD	C8B

# Primary Clustering

We have already observed the following phenomenon:

- With more insertions, the contiguous regions (or *clusters*) get larger

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
680						826	207	488	946		C8B	74C	ACD	3AD			946	B32							D59	5BA	19A	E9C			

The length of these chains will affect the number of probes required to perform insertions, accesses, or removals

# Primary Clustering

We currently have three clusters of length four

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
680						826	207	488	946		C8B	74C	ACD	3AD			946	B32								D59	5BA	19A	E9C			

Diagram illustrating the current state of a hash table with 32 slots (indices 0 to 1F). The table contains the following values:

Index	Value
0	680
1	
2	
3	
4	
5	
6	826
7	207
8	488
9	946
A	
B	C8B
C	74C
D	ACD
E	3AD
F	
10	
11	946
12	B32
13	
14	
15	
16	
17	
18	
19	D59
1A	5BA
1B	19A
1C	E9C
1D	
1E	
1F	

Three clusters of length four are identified by red double-headed arrows:

- Cluster 1: Indices 6, 7, 8, 9 (Values: 826, 207, 488, 946)
- Cluster 2: Indices 11, 12, 13, 14 (Values: 946, B32, , )
- Cluster 3: Indices 19, 1A, 1B, 1C (Values: D59, 5BA, 19A, E9C)

# Primary Clustering

There is a  $5/32 \approx 16\%$  chance that an insertion will fill A

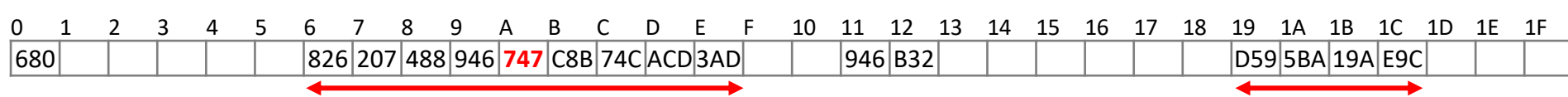
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	
680						826	207	488	946		C8B	74C	ACD	3AD			946	B32								D59	5BA	19A	E9C			

The diagram illustrates primary clustering in a hash table. Red double-headed arrows indicate the gaps between non-empty slots. The first arrow spans from index 6 to index 9. The second arrow spans from index 11 to index 14. The third arrow spans from index 19 to index 22.

# Primary Clustering

There is a  $5/32 \approx 16\%$  chance that an insertion will fill A


- This causes two clusters to *coalesce* into one larger cluster of length 9



# Primary Clustering

There is now a  $11/32 \approx 34\%$  chance that the next insertion will increase the length of this cluster

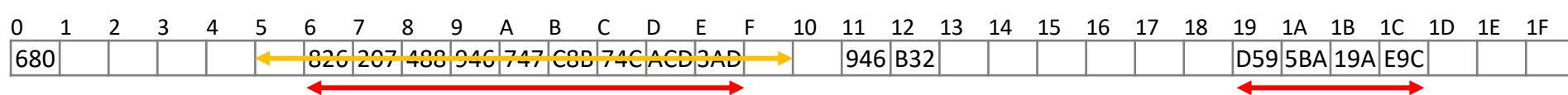
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
680						826	207	488	946	747	C8B	74C	ACD	3AD			946	B32							D59	5BA	19A	E9C			





# Primary Clustering

As the cluster length increases, the probability of further increasing the length increases



In general:

- Suppose that a cluster is of length  $\ell$
- An insertion either into any bin occupied by the chain or into the locations immediately before or after it will increase the length of the chain
- This gives a probability of  $\frac{\ell + 2}{M}$

# Run-time analysis

Therefore, we have three choices:

- Choose  $M$  large enough so that we will not pass this load factor
  - This could waste memory
- Double the number of bins if the chosen load factor is reached
- Choose a different strategy than linear probing
  - Two possibilities are quadratic probing and double hashing

# Outline

- Introduction
- Hash function
- Mapping down to  $0, \dots, M - 1$
- Dealing with collisions
  - Chained hash tables
  - Open addressing
    - Linear probing
    - Quadratic probing

# Description

Quadratic probing suggests moving forward by different amounts

For example,

```
int initial = hash_M( x.hash(), M );  
  
for ( int k = 0; k < M; ++k ) {  
    bin = (initial + k*k) % M;  
}
```

# Description

## Problem:

- Will `initial + k*k` step through all of the bins?
- Here, the array size is 10:

```
M = 10;
```

```
initial = 5
```

```
for ( int k = 0; k <= M; ++k ) {  
    std::cout << (initial + k*k) % M << ' '  
}
```

- The output is

```
5 6 9 4 1 0 1 4 9 6 5
```

# Description

## Problem:

- Will `initial + k*k` step through all of the bins?
- Now the array size is 12:

```
M = 12;
```

```
initial = 5
```

```
for ( int k = 0; k <= M; ++k ) {  
    std::cout << (initial + k*k) % M << ' '  
}
```

- The output is now

```
5 6 9 2 9 6 5 6 9 2 9 6 5
```

# Generalization

More generally, we could consider an approach like:

```
int initial = hash_M( x.hash(), M );  
  
for ( int k = 0; k < M; ++k ) {  
    bin = (initial + c1*k + c2*k*k) % M;  
}
```

## Using $M = 2^m$

If we ensure  $M = 2^m$  then choose

$$c_1 = c_2 = 1/2$$

```
int initial = hash_M( x.hash(), M );  
  
for ( int k = 0; k < M; ++k ) {  
    bin = (initial + (k + k*k)/2) % M;  
}
```

- Note that  $k + k*k$  is always even
- The growth is still  $\Theta(k^2)$
- This guarantees that all  $M$  entries are visited before the pattern repeats
  - This only works for powers of two



## Using $M = 2^m$

For example:

- Use an array size of 16:

```
M = 16;
```

```
initial = 5
```

```
for ( int k = 0; k <= M; ++k ) {  
    std::cout << (initial + (k + k*k)/2) % M << ' '  
}
```

- The output is now

```
5 6 8 11 15 4 10 1 9 2 12 7 3 0 14 13 13
```

# Example

Consider a hash table with  $M = 16$  bins

Given a 2-digit hexadecimal number:

- The least-significant digit is the primary hash function (bin)
- Example: for  $7A_{16}$ , the initial bin is  $A$

# Example

Insert these numbers into this initially empty hash table

9A, 07, AD, 88, BA, 80, 4C, 26, 46, C9, 32, 7A, BF, 9C

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

# Example

Next we must insert BA

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							07	88		9A			AD		

# Example

Next we must insert BA

- The next bin is empty

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
							07	88		9A	BA		AD		

# Example

Next we are adding 80, 4C, 26

- All the bins are empty—simply insert them

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88		9A	BA	4C	AD		

# Example

Next, we must insert 46

- Bin **6** is occupied
- Bin **6 + 1 = 7** is occupied
- Bin **7 + 2 = 9** is empty

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88	46	9A	BA	4C	AD		

# Example

Next, we must insert C9

- Bin **9** is occupied
- Bin **9 + 1 = A** is occupied
- Bin **A + 2 = C** is occupied
- Bin **C + 3 = F** is empty

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80						26	07	88	46	9A	BA	4C	AD		C9



# Example

Next, we insert 7A

- Bin **A** is occupied
- Bins **A + 1 = B**, **B + 2 = D** and **D + 3 = 0** are occupied
- Bin **0 + 4 = 4** is empty

0	1	2	3	4	5	6	7	8	9	<b>A</b>	B	C	D	E	F
80		32		<b>7A</b>		26	07	88	46	9A	BA	4C	AD		C9

# Example

Having completed these insertions:

- The load factor is  $\lambda = 14/16 = 0.875$
- The average number of probes is  $32/14 \approx 2.29$

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80	9C	32		7A	BF	26	07	88	46	9A	BA	4C	AD		C9

# Erase

Can we erase an object like we did with linear probing?

- Consider erasing 9A from this table
- There are  $M - 1$  possible locations where an object which could have occupied a position could be located

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80	21		43			76				9A					50

Instead, we use *lazy erasing*

- Mark a bin as ERASED; however, when searching, treat the bin as occupied and continue

# Summary

In this topic, we have looked at quadratic probing:

- An open addressing technique
- Steps forward by a quadratically growing steps
- Insertions and searching are straight forward
- Removing objects is more complicated: use lazy deletion
- Still subject to secondary probing

# Summary

