CSCE 4263/5183
Advanced Data Structures

Lecture 15

Hashing Topics (continue)

Fall 2025

Prof. Khoa Luu
khoaluu@uark.edu

Notice

- HW 4 – Submission deadline: Nov. 11, 2025
- Final Project – Q&A

Outline

Previously, we considered means for calculating 32-bit hash values

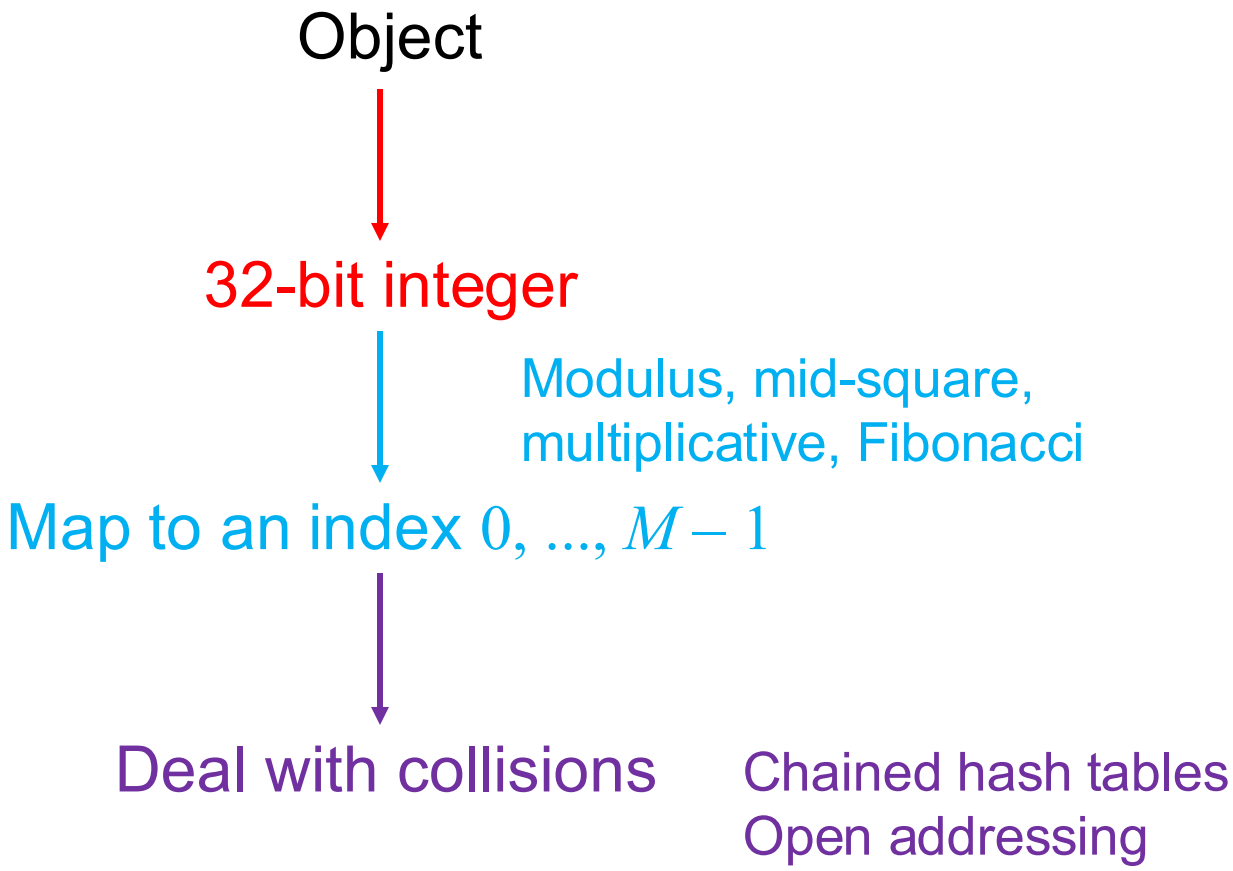
- Explicitly defined hash values
- Implicitly calculated hash values

Practically, we will require a hash value on the range $0, ..., M-1$:

- The modulus operator %
- Review of bitwise operations
- The multiplicative method

9.3.1

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

- We can simplify this if $M = 2^m$
- We can use logic operations
  - We will review 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:

0000025 and 0000325

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:

`00000000`0101 and `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: `100011001010000`

Similarly, 100011100101_2 / 1000000_2 == 100011

The bitwise operators: & << >>

This can be done mechanically by shifting the bits appropriately:

$$1000\,1110\,0101_2 \texttt{ << } 4 \texttt{ == } 1000\,1110\,0101\,0000_2$$

$$1000\,1110\,0101_2 \texttt{ >> } 6 \texttt{ == } 10\,0011_2$$

Powers of 2 are now easy to calculate:

$$1_2 \texttt{ << } 4 \texttt{ == } 1\,0000_2 \qquad \texttt{// } 2^4 = 16$$

$$1_2 \texttt{ << } 6 \texttt{ == } 100\,0000_2 \qquad \texttt{// } 2^6 = 64$$

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
- Thus, if $x \neq y$, $h(x) \neq h(y)$
- However, the probability that $h_M(\_h(x)\_) = h_M(\_h(y)\_)$ is one in $M/8$
  - This is not one in M

Modulo a power of two

For some objects, it is worse:

- Instance of `Single_node<int>` on linux always have `10000` as the last five bits
  - This increases the probability of a collision to one in $M/32$

Fortunately, the multiplicative method resolves this issue

Multiplication techniques

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:

```
    10100111
  × 11010011
  ----------
    10100111
   10100111
  10100111
 10100111
10100111
 10100111
+10100111
----------------
1000101110100101
```

C++ truncates the product of two n-bit numbers to n bits

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

10100011 × 11010011
= 1000011001011001

Multiplication techniques

Multiplication can be used in encryption as with IDEA:

```
unsigned short a, b, c, d; // 64 bit number broken into four 16 bits
unsigned short z[9][6];    // 52 numbers based on key (2 unused)

for ( int i = 0; i < 8; ++i ) {
    a *= z[i][0];  b += z[i][1];  c += z[i][2];  d *= z[i][3];
    e = z[i][4]*(a^c);
    f = ( e + (b^d) )*z[i][5];
    e += f;
    a ^= f;    tmp = b;    b = c^f;    c = tmp^e;    d ^= e;
}

 a *= z[8][0];  b += z[8][1];  c += z[8][2];  d *= z[8][3];
```

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

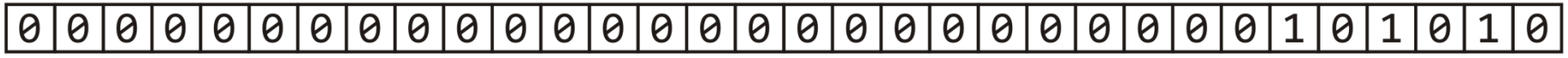
The multiplicative method

$m = 10$
$n = 42$

Next, $n = 42$ or 101010_2

```
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

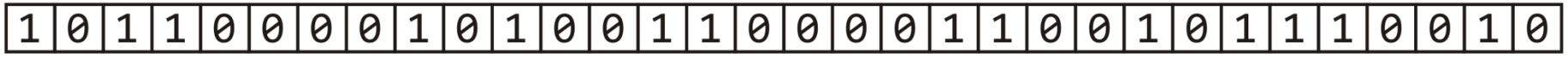
$m = 10$

$n = 42$

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



The multiplicative method

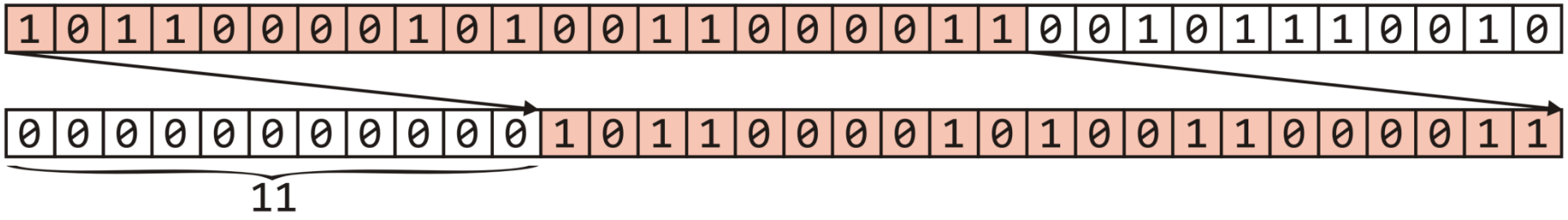
$m = 10$

$n = 42$

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

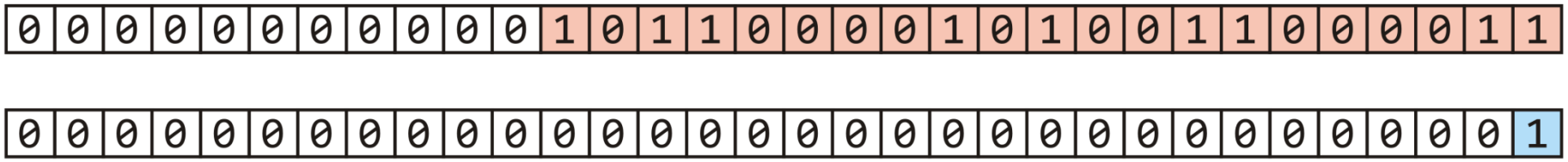


The multiplicative method

$m = 10$
$n = 42$

Next, start with 1

```
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);
}
```



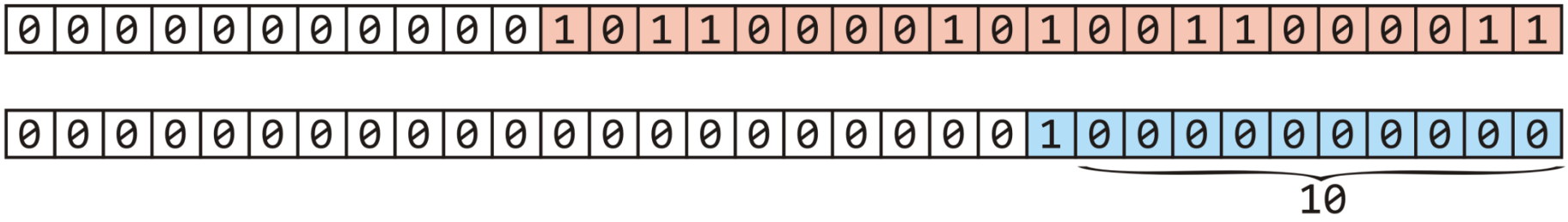
The multiplicative method

$m = 10$

$n = 42$

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);
}
```

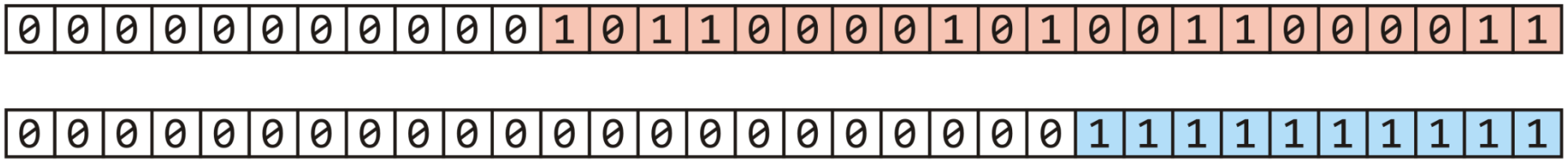


The multiplicative method

$m = 10$
$n = 42$

Subtracting 1 yields $m = 10$ ones

```
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);
}
```

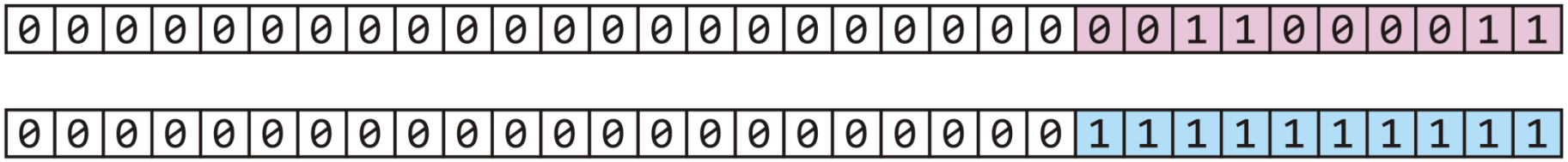


The multiplicative method

$m = 10$
$n = 42$

Taken the bitwise to clear all but the last 10 bits

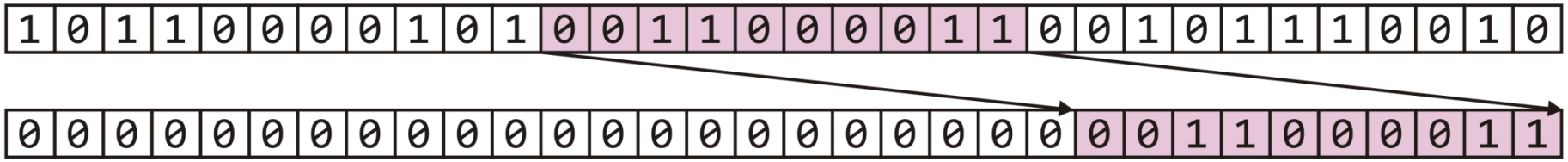
```
const unsigned int C = 581869333;  // some number

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

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

```
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);
}
```



$$h_M(42) = 195$$

Dealing with signed integers

In some cases (Java) unsigned integers are either not available

Our function would be implemented as follows:

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

Is this sufficient to yield a value on $0, ..., M-1$?

Dealing with signed integers

The modulus operator % satisfies the following equality:

```
n == M*(n/M) + n%M
```

The division operator n/M truncates any factional part

- It rounds towards zero

For positive n and M, this yields positive values:

- If `n = 13` and `M = 8`, `13/8 → 1` and `8*1 → 8`
  ∴ `13 % 8 → 5`

For a negative value of n:

- If `n = -11` and `M = 8`, `-11/8 → -1` and `8*(-1) = -8`
  ∴`(-11) % 8 → -3`

Dealing with signed integers

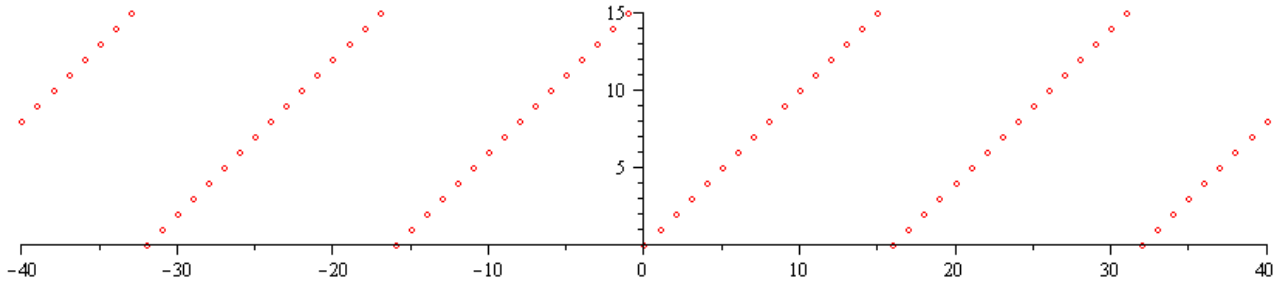
Our mapping function could be implemented as follows:

```
int hash_M( int n, int M ) {
    int hash_value = n % M;
    return (hash_value >= 0) ? hash_value : hash_value + M;
}
```

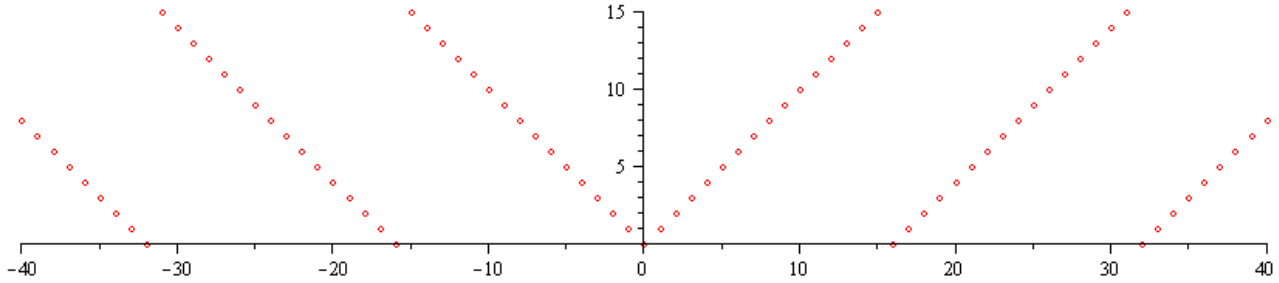
Dealing with signed integers

Why `+ M` and not `std::abs( M )`?

- First, adding results in periodicity:



- Using the absolute value is not periodic



Dealing with signed integers

Also, not all integers have absolute values:

```
int main() {
    int n1 = -2147483648;  // -2^31
    int n2 = -2147483647;  // -(2^31 – 1)

        cout << std::abs( n1 ) << endl;
        cout << std::abs( n2 ) << endl;

    return 0;
}
```

The output is:

```
-2147483648
 2147483647
```

Summary

This topic covered next step in creating a hash value:

- Taking a 32-bit number and mapping it down to $0, \ldots, M-1$
- One can use the modulus; however, using the middle m bits of the number times a large prime avoids some weaknesses of the modulus

We must now deal with *collisions*: the reality that two objects may hash to the same value

Chained Hash Tables

We have:

- Discussed techniques for hashing
- Discussed mapping down to a given range $0, ..., M-1$

Now we must deal with collisions

- Numerous techniques exist
- Containers in general
  - Specifically linked lists

Background

First, a review:

- We want to store objects in an array of size M
- We want to quickly calculate the bin where we want to store the object
  - We came up with hash functions—hopefully $\Theta(1)$
  - Perfect hash functions (no collisions) are difficult to design
- We will look at some schemes for dealing with collisions

Implementation

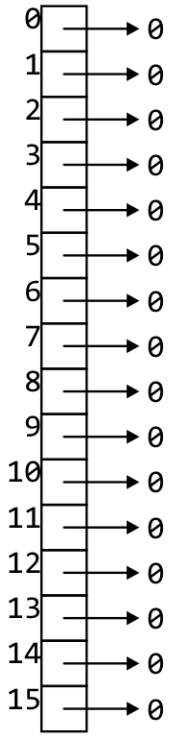
Consider associating each bin with a linked list:

```
template <class Type>
class Chained_hash_table {
    private:
        int table_capacity;
        int table_size;
        Single_list<Type> *table;

        unsigned int hash( Type const & );

    public:
        Chained_hash_table( int = 16 );
        int count( Type const & ) const;
        void insert( Type const & );
        int erase( Type const & );
        // ...
};
```

Implementation

The constructor creates an array of n linked lists

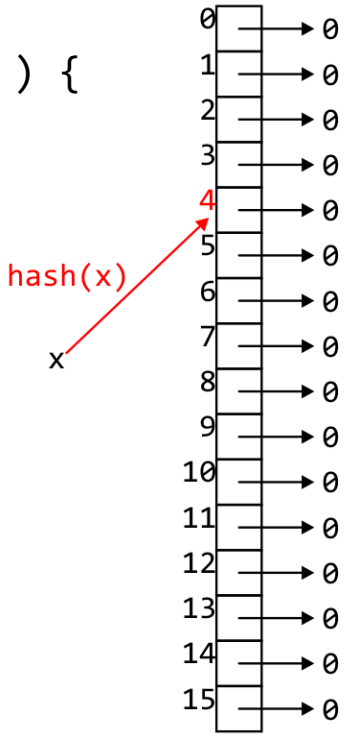
```
template <class Type>
Chained_hash_table::Chained_hash_table( int n ):
table_capacity( std::max( n, 1 ) ),
table_size( 0 ),
table( new Single_list<Type>[table_capacity] ) {
    // empty constructor
}
```



Implementation

The function hash will determine the bin of an object:

```
template <class Type>
int Chained_hash_table::hash( Type const &obj ) {
    return hash_M( obj.hash(), capacity() );
}
```



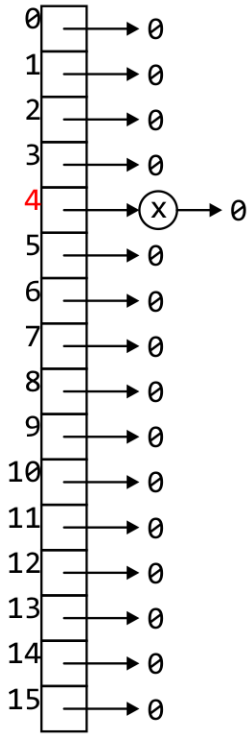
Recall:

- `obj.hash()` returns a 32-bit non-negative integer
- `unsigned int hash_M( obj, M )` returns a value in $0, \ldots, M-1$

Implementation

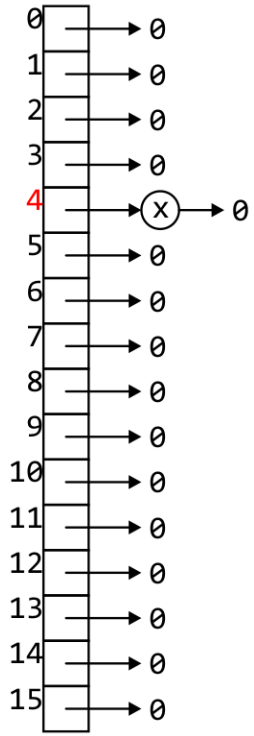
Other functions mapped to corresponding linked list functions:

```
template <class Type>
void Chained_hash_table::insert( Type const &obj ) {
    unsigned int bin = hash( obj );

    if ( table[bin].count( obj ) == 0 ) {
        table[bin].push_front( obj );
        ++table_size;
    }
}
```



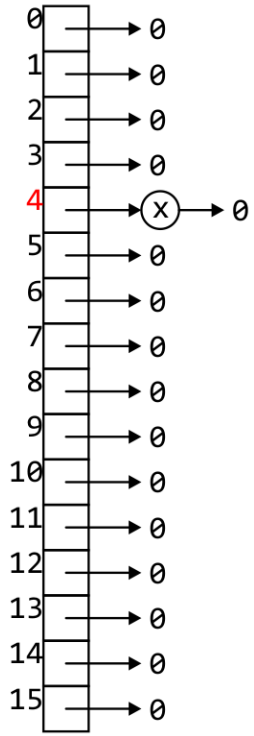
Implementation

```
template <class Type>
int Chained_hash_table::count( Type const &obj ) const {
    return table[hash( obj )].count( obj );
}
```



Implementation

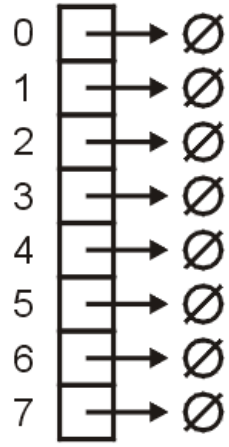
```
template <class Type>
int Chained_hash_table::erase( Type const &obj ) {
    unsigned int bin = hash( obj );

    if ( table[bin].erase( obj ) ) {
        --table_size;
        return 1;
    } else {
        return 0;
    }
}
```



Example

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

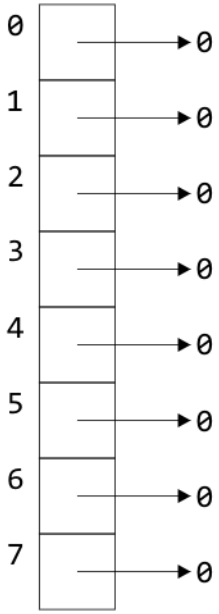
- We will choose the bin based on the host name
- Associated with the name will be the IP address
- *E.g.*, `("optimal", 129.97.94.57)`



Example

We will store strings and the hash value of a string will be the last 3 bits of the first character in the host name

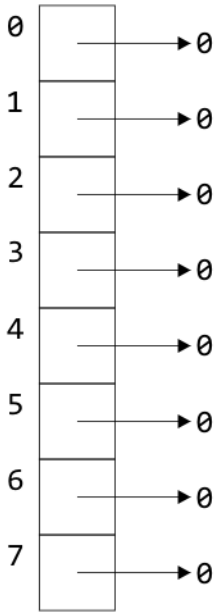
- The hash of `"optimal"` is based on `"o"`



Example

The following is a list of the binary representation of each letter:

- "a" is 1 and it cycles from there…

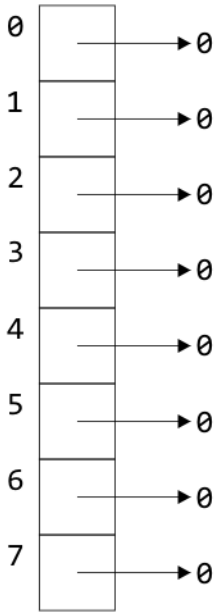


a	01100001	n	01101110
b	01100010	o	01101111
c	01100011	p	01110000
d	01100100	q	01110001
e	01100101	r	01110010
f	01100110	s	01110011
g	01100111	t	01110100
h	01101000	u	01110101
i	01101001	v	01110110
j	01101010	w	01110111
k	01101011	x	01111000
l	01101100	y	01111001
m	01101101	z	01111010

Example

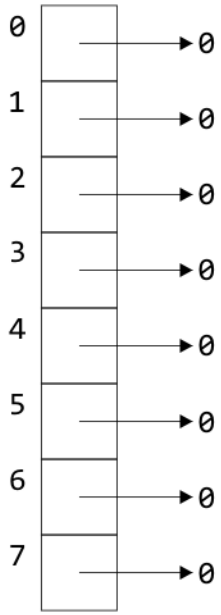
Our hash function is

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

     return str[0] & 7;
}
```



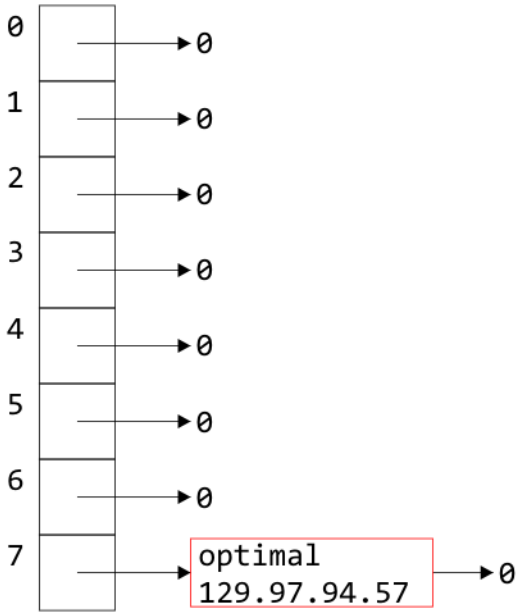
Example

Starting win an array of 8 empty linked lists



Example

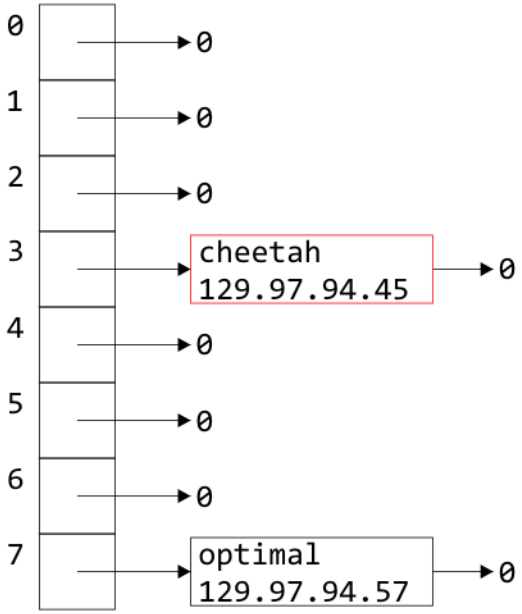
The pair `("optimal", 129.97.94.57)` is entered into bin `01101111` = 7



Example

Similarly, as `"c"` hashes to 3

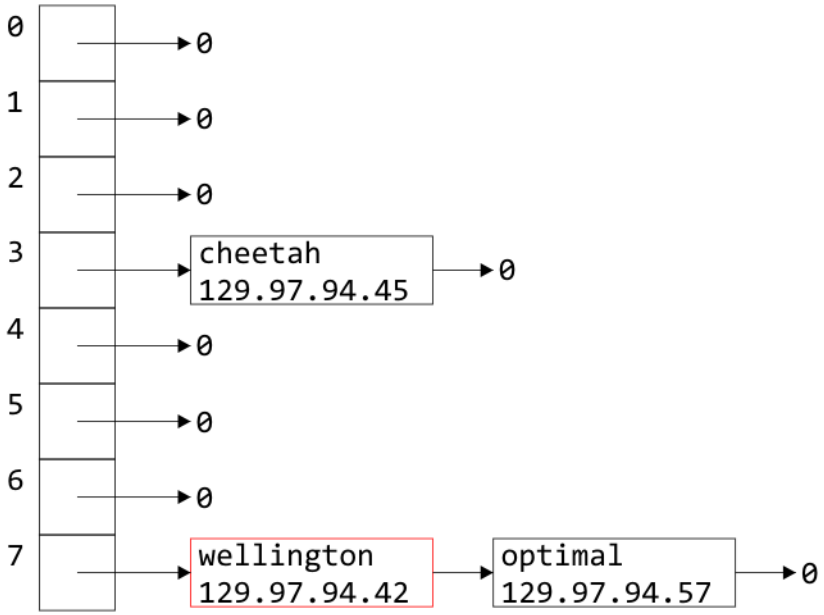
- The pair `("cheetah", 129.97.94.45)` is entered into bin 3



Example

The "w" in Wellington also hashes to 7

- `("wellington", 129.97.94.42)` is entered into bin 7



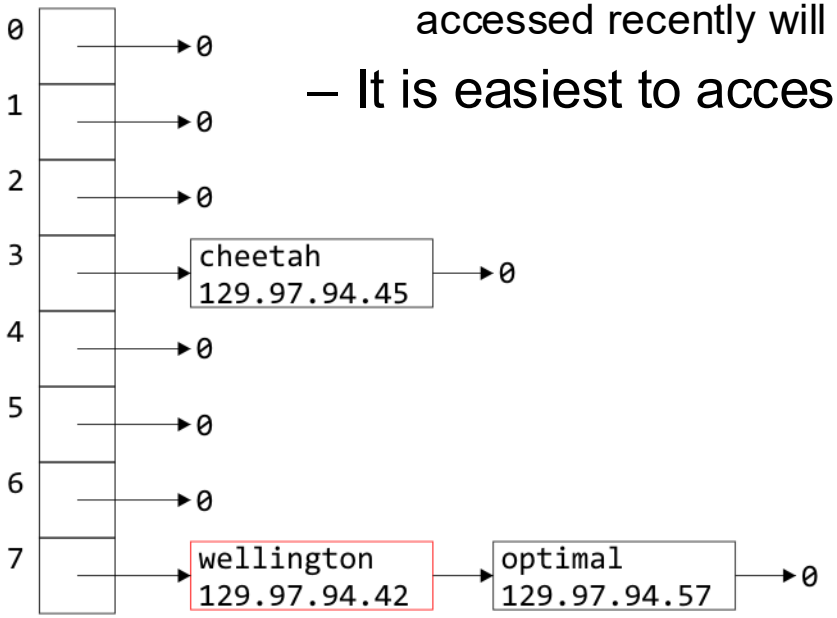
Example

Why did I use `push_front` from the linked list?

- Do I have a choice?
- A good heuristic is

  "unless you know otherwise, data which has been accessed recently will be accessed again in the near future"

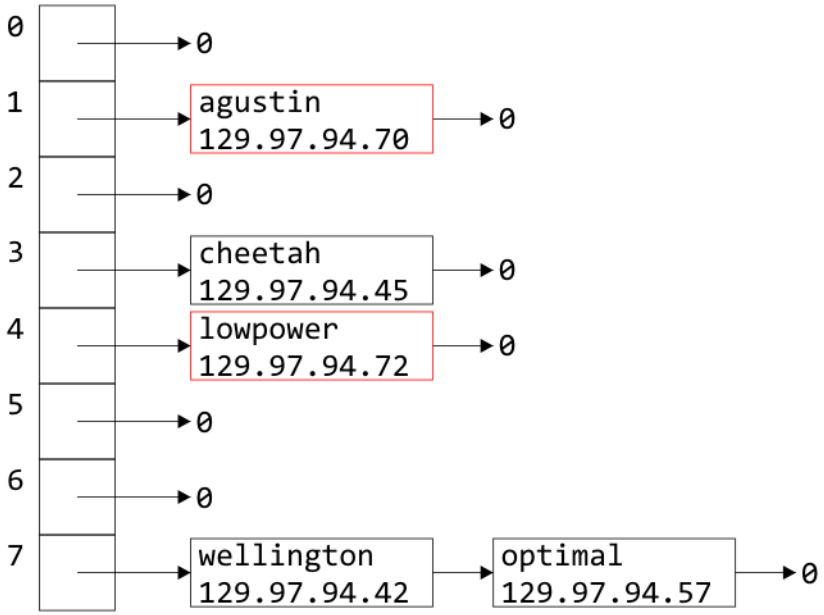
  - It is easiest to access data at the front of a linked list



Heuristics include rules of thumb, educated guesses, and intuition

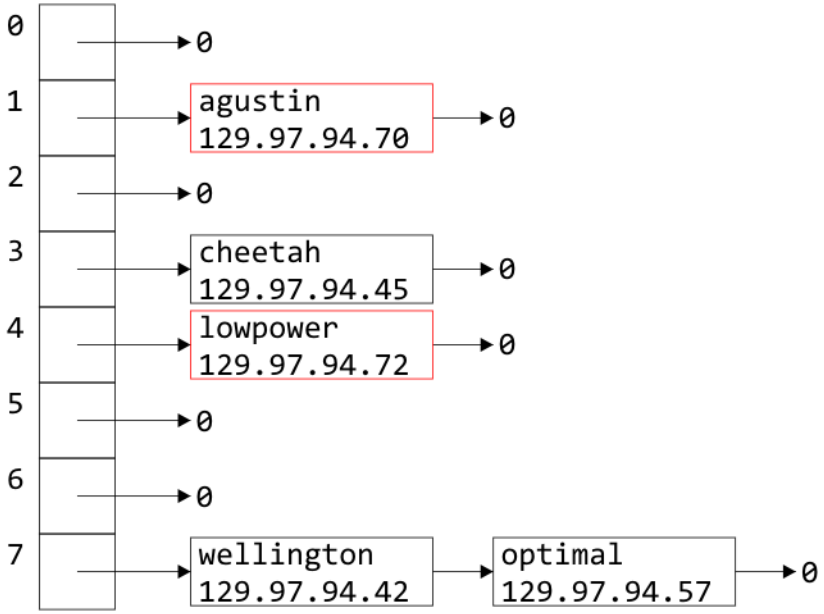
Example

Similarly we can insert the host names `"augustin"` and `"lowpower"`



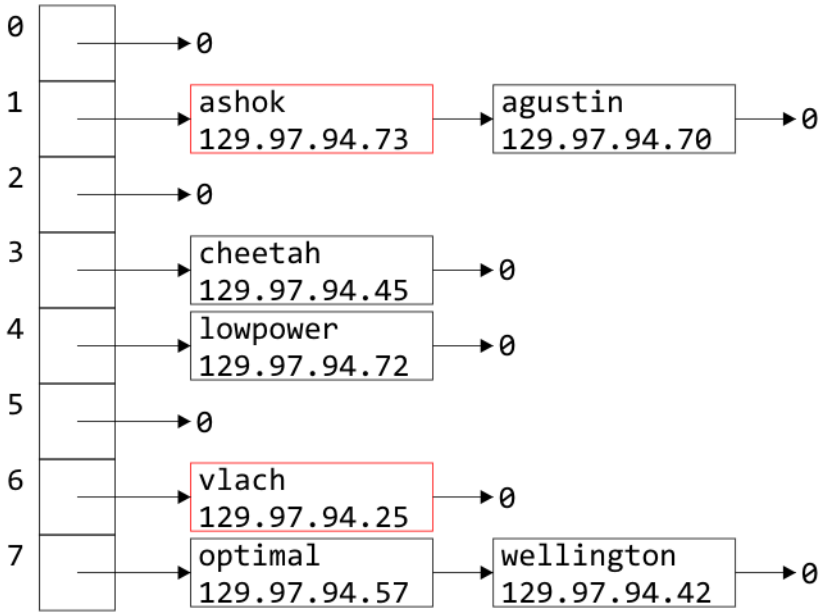
Example

If we now wanted the IP address for `"optimal"`, we would simply hash `"optimal"` to 7, walk through the linked list, and access 129.97.94.57 when we access the node containing the relevant string



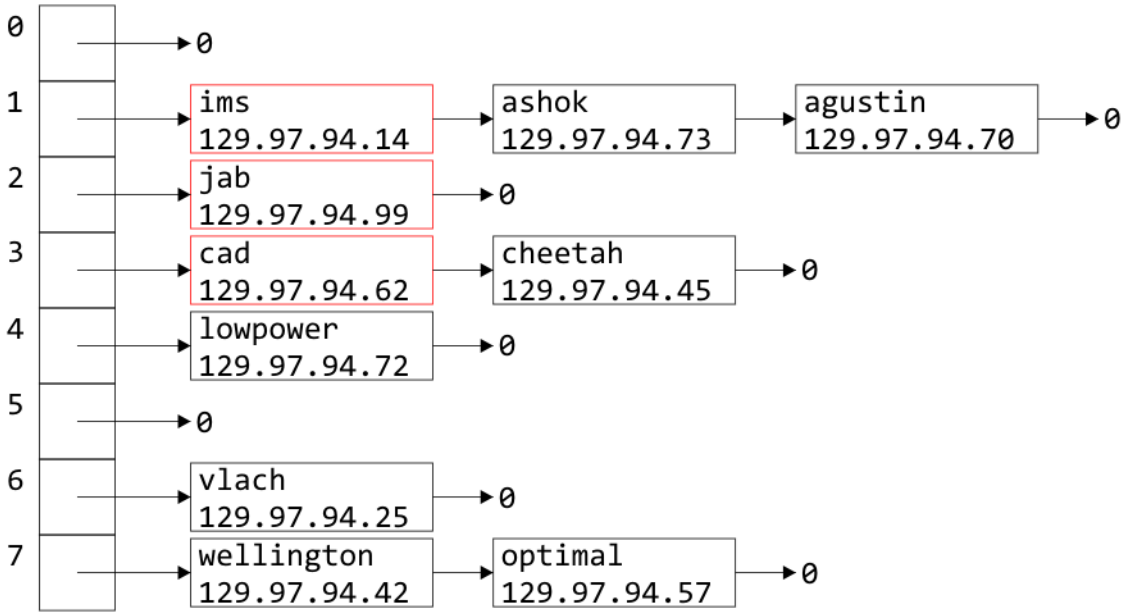
Example

Similarly, "ashok" and "vlach" are entered into bin 7



Example

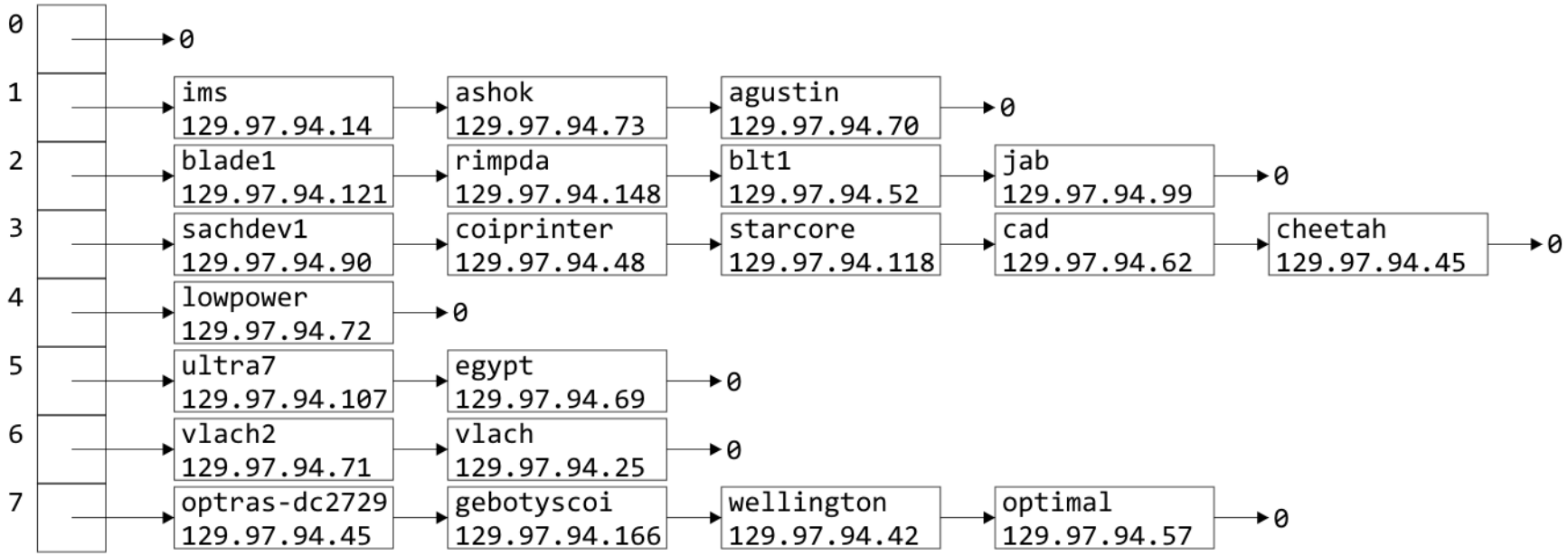
Inserting `"ims"`, `"jab"`, and `"cad"` doesn't even out the bins



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 $\mathbf{O}(k)$



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: $\mathbf{O}(\lambda)$

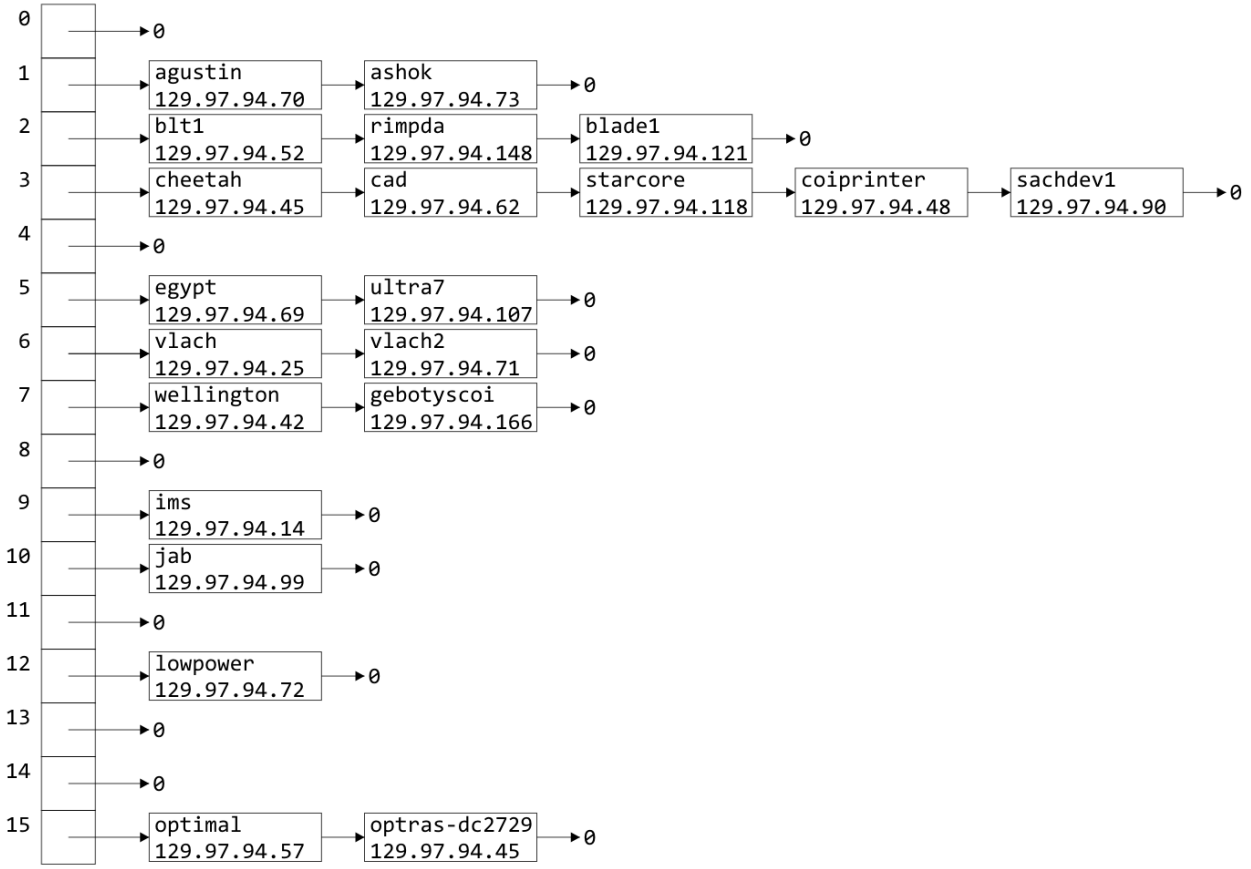
The most obvious solution is to double the size of the hash table

- Unfortunately, the hash function must now change
- In our example, the doubling the hash table size requires us to take, for example, the last four bits

Doubling Size

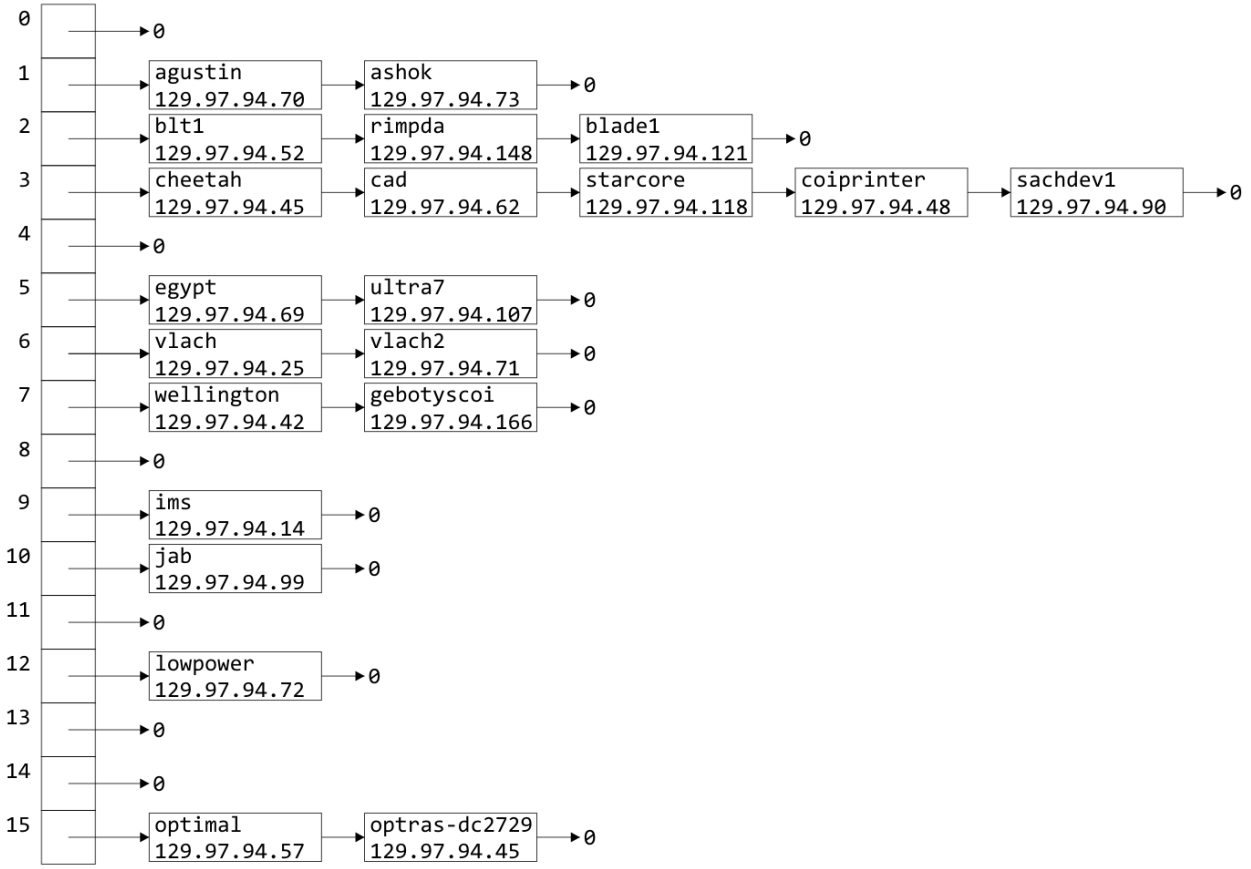
The load factor is now $\lambda = 1.3125$

- 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

- Any hash function based on anything other than every letter will cause clustering

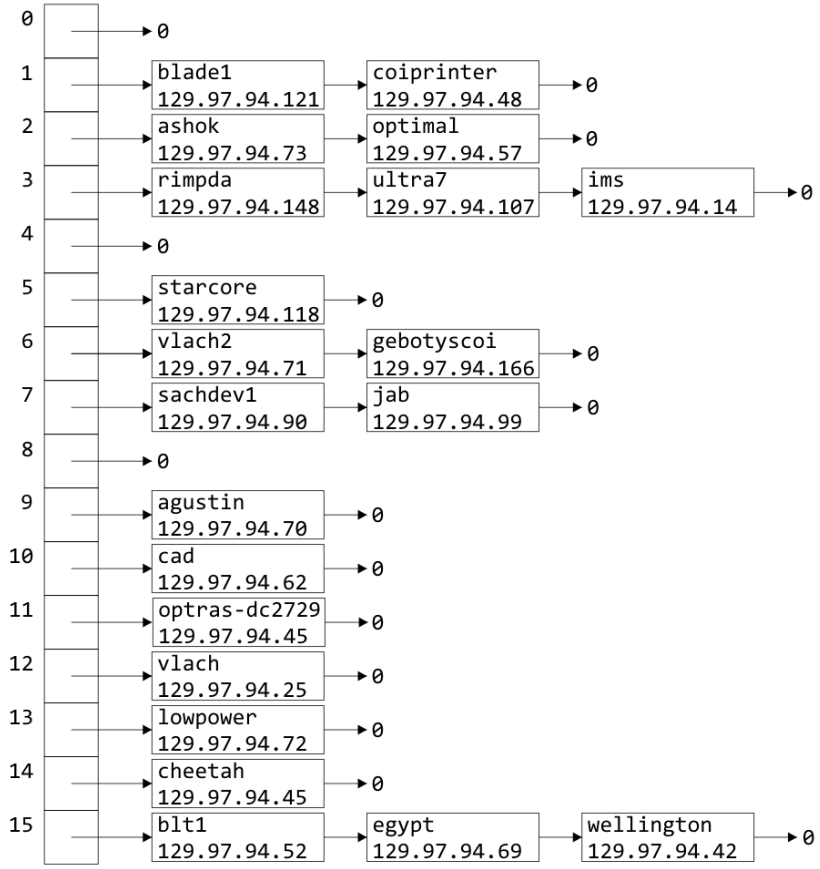
Choosing a Good Hash Function

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

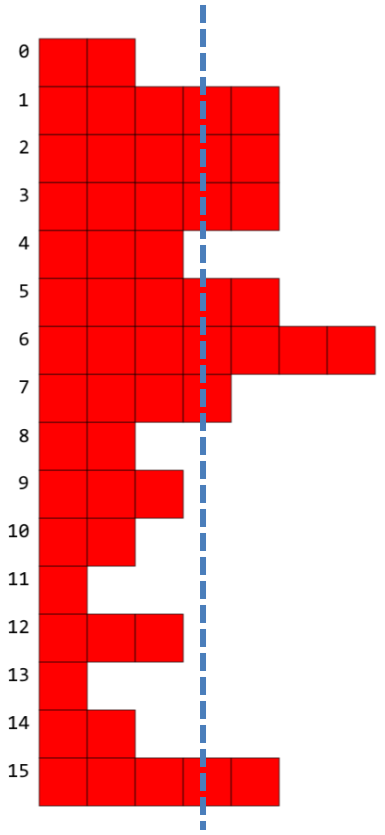
This hash function yields a much nicer distribution:



Choosing a Good Hash Function

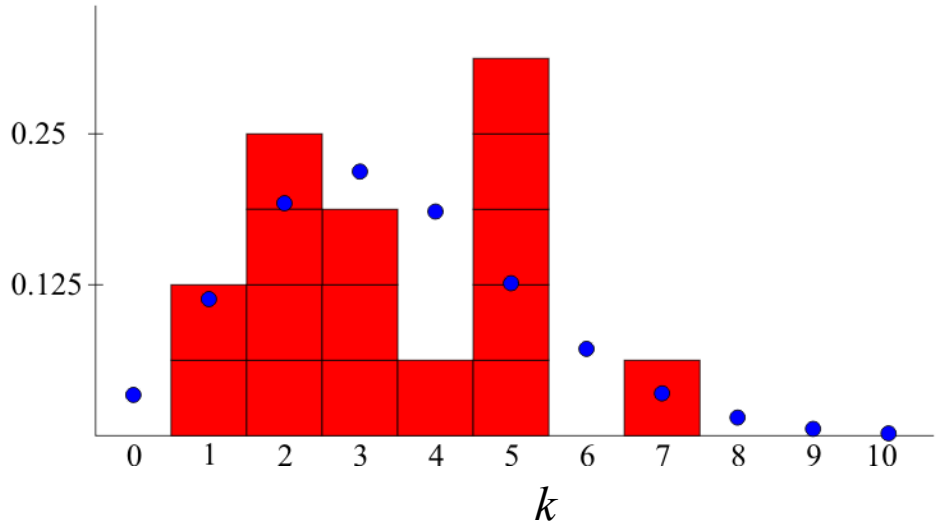
When we insert the names of all 55 names, we would have a load factor $\lambda = 3.4375$

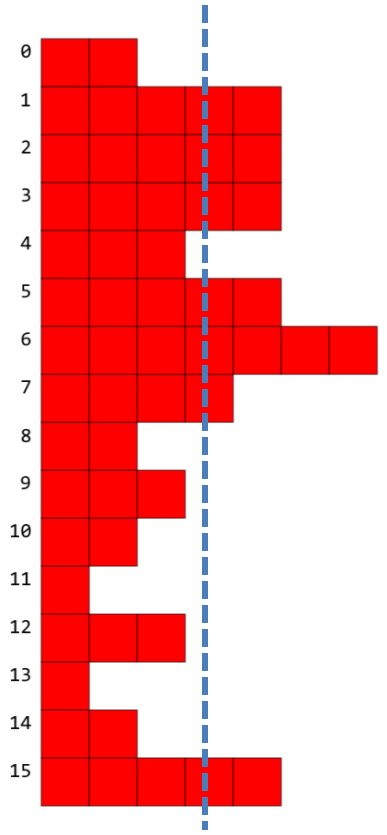
- Clearly there are not exactly 3.4375 objects per bin
- How can we tell if this is a good hash function?
- Can we expect exactly 3.4375 objects per bin?
  - Clearly no…
- The answer is with statistics…



Choosing a Good Hash Function

We would expect the number of bins which hold k objects to approximately follow a Poisson distribution





Problems with Linked Lists

One significant issue with chained hash tables using linked lists

- It requires extra memory
- It uses dynamic memory allocation

Total memory:

16 bytes

- A pointer to an array, initial and current number of bins, and the size

$+ 12M$ bytes ($8M$ if we remove count from `Single_list`)

$+ 8n$ bytes if each object is 4 bytes

Problems with linked lists

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

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

We could look at other techniques:

- Scatter tables: use other bins and link the bins
- Use an alternate memory allocation model

Black Board Example

Use the hash function

```
unsigned int hash( unsigned int n ) { return n % 10; }
```

to enter the following 15 numbers into a hash table with 10 bins:

534, 415, 465, 459, 869, 442, 840, 180, 450, 265, 23, 946, 657, 3, 29

Summary

The easiest way to deal with collisions is to associate each bin with a container

We looked at bins of linked lists

- The example used host names and IP addresses
- We defined the load factor $\lambda = n/M$
- Discussed doubling the number of bins
- Our goals are to choose a good hash function and to keep the load factor low
- We discussed alternatives

Next we will see a different technique using only one array of bins: open addressing