An introduction to hash tables

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

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….

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()'
```

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…

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

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

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 $10000001\ 01100001\ 00001010\ 10110011_2$
- This can be written as `http://129.97.10.179/`
- We use domain names because IP addresses are not human readable

IP Addresses

Similarly, the University of Waterloo has 2^{16} IP addrescontrol of names within its domain

- Any IP address starting with `129.97` belongs to UW
- This gives UW 256^2 = 65535 IP addresses

The University of Waterloo currently uses ~60 % of the IP addresses

IP Addresses

Given an IP address, if we wanted to quickly find any associated domain name, we could create an array of size 65536 of strings:

```
int const MAX_IP_ADDRESSES = 65536
string domain_name[MAX_IP_ADDRESSES];
```

For example, my computer is churchill.uwaterloo.ca and its IP address is `http://129.97.10.179/`

- The prefix 129.97 is common to all uWaterloo IP addresses
- As $179 + 256 \times 10 = 2739$, it follows that

```
domain_name[2739] = "churchill";
```

IP Addresses

Under IPv6, IP addresses are 128 bits

- It combines what is now implemented as subnets as well as allowing for many more IP addresses

Suppose uWaterloo is allocated 2^{64} IP addresses under this scheme

- We cannot allocate an array of size 2^{64}

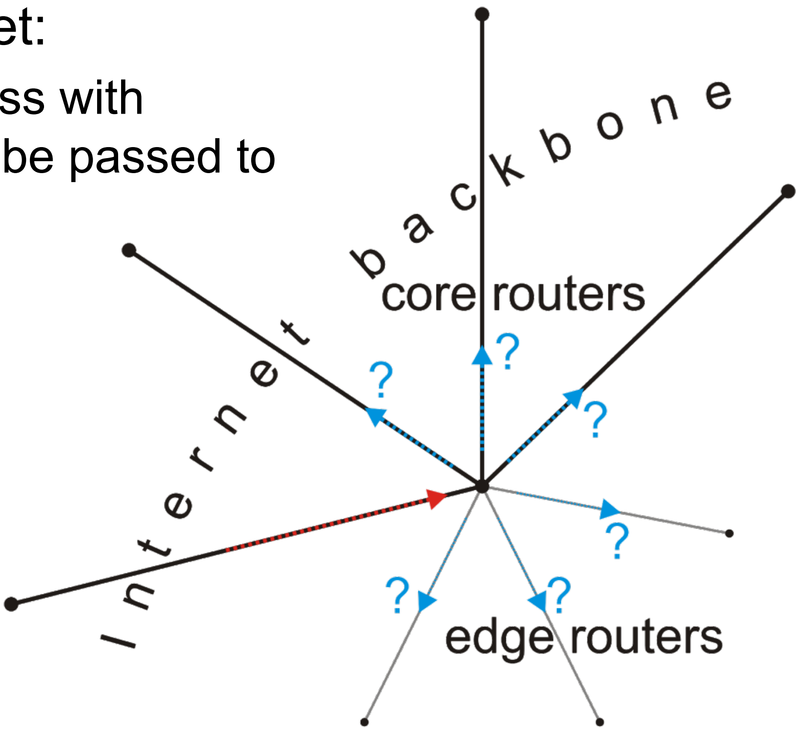
IP Addresses

What if we want to associate domain names with IP addresses?

- Which entry in the array should be associated with the domain name `churchill.uwaterloo.ca` ?

Consider core routers on the Internet:

- They must associate each IP address with the next router that address should be passed to



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?
- Suppose Jane Doe has the number 20123456
- I can't 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?
- First three digits might cause a problem: almost all students start with 201, 202, 203, 204, or 205
- The last three digits, however, are essentially random

Therefore, I could store Jane's examination grade

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

Simpler problem

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

Simpler problem

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

- I can store something in that location
- Storing it, accessing it, and erasing it is Θ(1)
- Problem: two or more students may map to the same number:
  - Wěi Wang has ID 20173456 and scored 85
  - Alma Ahmed has ID 2024456 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

IP Addresses

Going back to the problem with IP addresses

- We need a hash function to map an IP address to a smaller range
- We need a hash function to map a domain name to a smaller range

Mapping `129.97.10.179` onto a smaller range may seem easier, but a mechanism for mapping churchill.uwaterloo.ca onto a small range of integers may be more interesting

The hash process

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

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

$$\begin{tikzcd}
\text{Object} \arrow[d, "\text{Techniques vary...}"] \\
\text{32-bit integer} \arrow[d, "\text{Modulus, mid-square, multiplicative, Fibonacci}"] \\
\text{Map to an index } 0, \ldots, M-1 \arrow[d] \\
\text{Deal with collisions} & \text{Chained hash tables} \\
& \text{Open addressing} \arrow[dl] \\
\text{Linear probing} \\
\text{Quadratic probing} \\
\text{Double hashing}
\end{tikzcd}$$

Summary

Discuss storing unordered data

Discuss IP addresses and domain names

Consider conversions between these two forms

Introduce the idea of using a smaller array

- Converted “large” numbers into valid array indices
- Reduces $\mathbf{O}(\ln(n))$ in arrays and AVL trees to to $\mathbf{O}(1)$

Discussed the issues with collisions

References

Wikipedia, http://en.wikipedia.org/wiki/Hash_table

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

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

Hash functions

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`

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

$$\begin{tikzcd}
\text{Object} \arrow[d] \\
\text{32-bit integer} \arrow[d] & \text{Modulus, mid-square, multiplicative, Fibonacci} \\
\text{Map to an index } 0, \ldots, M-1 \arrow[d] \\
\text{Deal with collisions} & \text{Chained hash tables} \\
& \text{Open addressing} \arrow[dl] \\
\text{Linear probing} \\
\text{Quadratic probing} \\
\text{Double hashing}
\end{tikzcd}$$

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

Types of hash functions

We will look at two classes of hash functions

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

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;
}
```

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

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

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

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 );
}
```

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

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

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 );
}
```

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 );
}
```

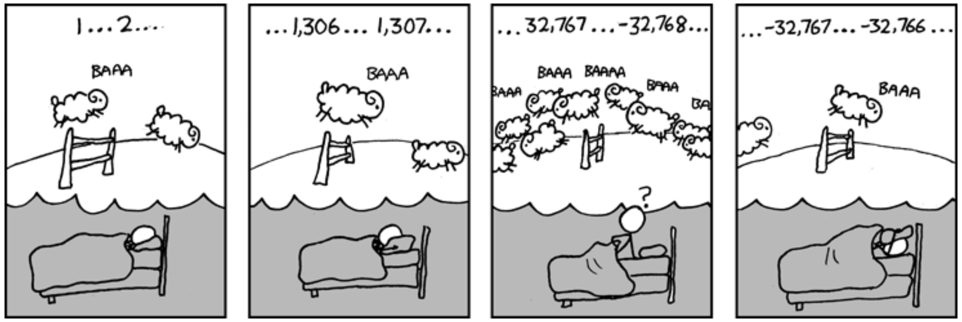
Rational number class

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

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

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
```

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;
    }
}
```

Rational number class

Problem:

– The rational numbers $\frac{1}{2}$ and $-\frac{1}{2}$ (negative) have different values

– The output of

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

is

```
858993503
3435973793
```

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

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;
}
```

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 $0, 1, ..., M - 1$

References

Wikipedia, http://en.wikipedia.org/wiki/Hash_function

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

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

Mapping down to $0, \ldots, M-1$

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 hash process

$$\begin{tikzcd}[row sep=large]
\text{Object} \arrow[d] & \\
\text{32-bit integer} \arrow[d] & \text{Modulus, mid-square, multiplicative, Fibonacci} \\
\text{Map to an index } 0, \ldots, M-1 \arrow[d] & \\
\text{Deal with collisions} & \text{Chained hash tables} \\
& \text{Open addressing}
\end{tikzcd}$$

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:

`000000000101` and `000000100101`

The bitwise operators: & << >>

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

$\texttt{1000 1110 0101}_2$ & $\texttt{0000 0000 1111}_2 \rightarrow \texttt{0000 0000 0101}_2$

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

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:

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

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

Powers of 2 are now easy to calculate:

1_2 << 4 == $1\ 0000_2$ // $2^4 = 16$

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

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 $h_M(x) = h_M(y)$ 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 ecelinux 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

Chained hash tables

Outline

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

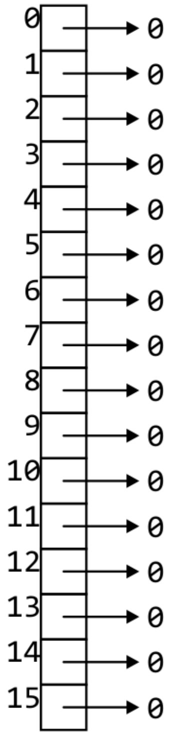
Consider associating each bin with a linked list:

```
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

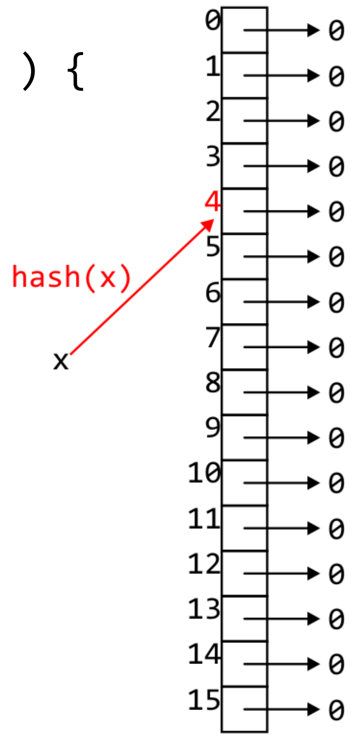
```
template <class Type>
Chained_hash_table::Chained_hash_table( int n ):
table_capacity( std::max( n, 1 ) ),
table_size( 0 ),
table( new Single_list<Type>[table_capacity] ) {
    // empty constructor
}
```



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() );
}
```



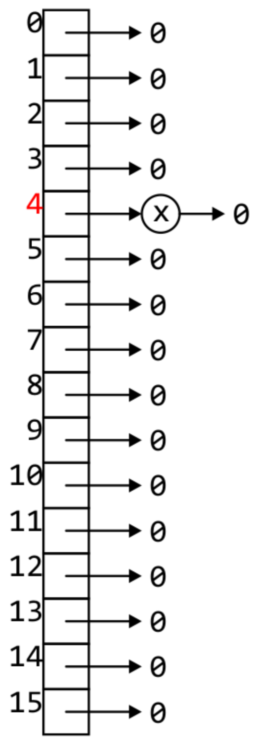
Recall:

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

Implementation

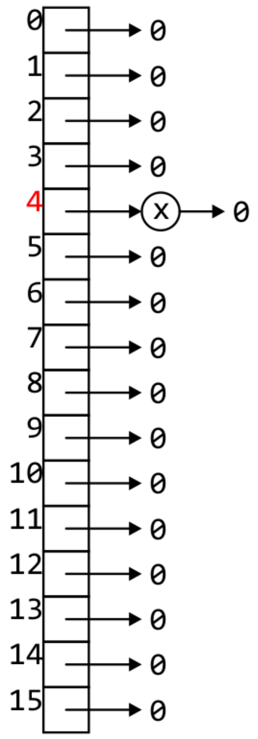
Other functions mapped to corresponding linked list functions:

```
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;
    }
}
```



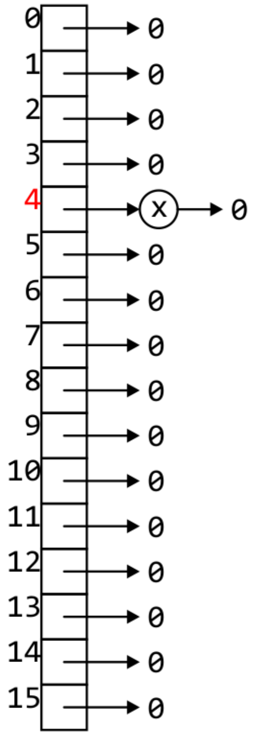
Implementation

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



Implementation

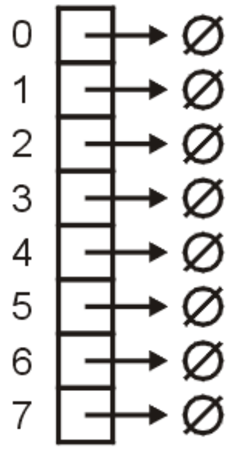
```
template <class Type>
int Chained_hash_table::erase( Type const &obj ) {
    unsigned int bin = hash( obj );

    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

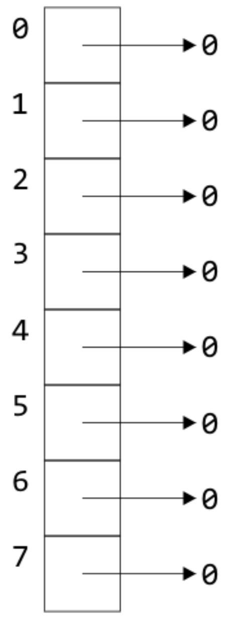
- We will choose the bin based on the host name
- Associated with the name will be the IP address
- *E.g.*, `("optimal", 129.97.94.57)`



Example

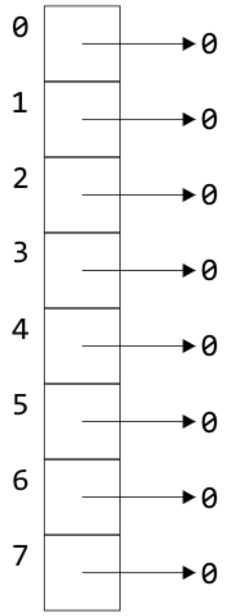
We will store strings and the hash value of a string will be the last 3 bits of the first character in the host name

- 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…

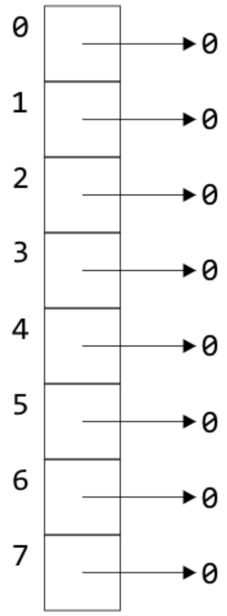


a	01100001	n	01101110
b	01100010	o	01101111
c	01100011	p	01110000
d	01100100	q	01110001
e	01100101	r	01110010
f	01100110	s	01110011
g	01100111	t	01110100
h	01101000	u	01110101
i	01101001	v	01110110
j	01101010	w	01110111
k	01101011	x	01111000
l	01101100	y	01111001
m	01101101	z	01111010

Example

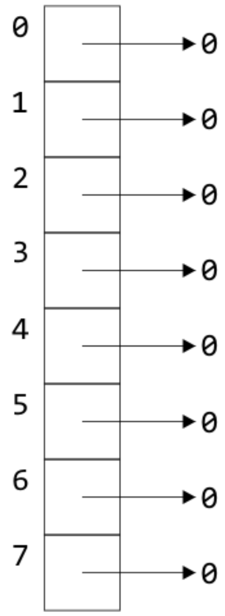
Our hash function is

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

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



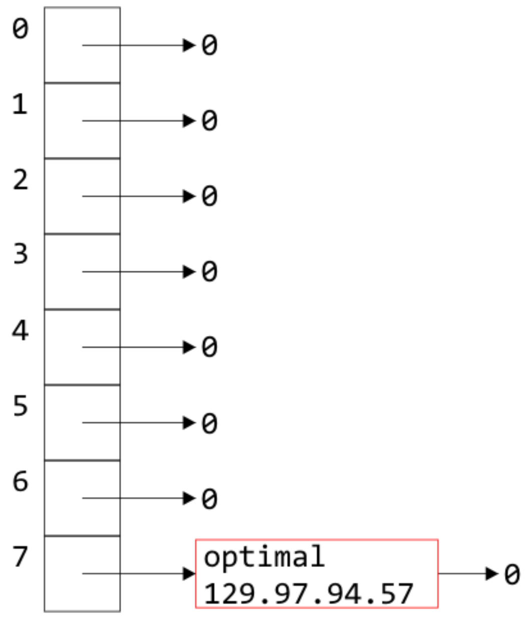
Example

Starting win an array of 8 empty linked lists



Example

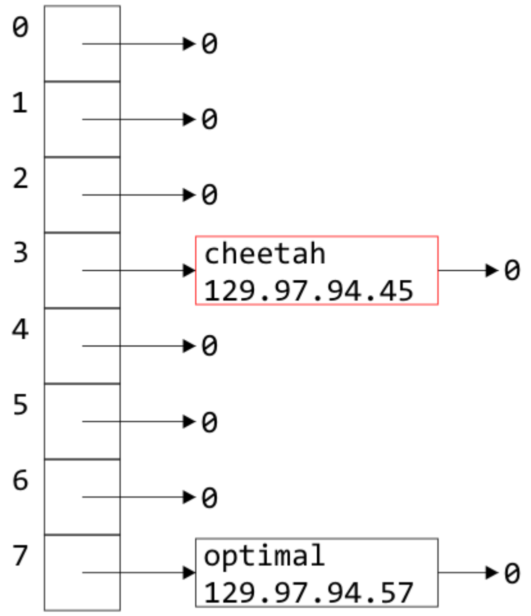
The pair `("optimal", 129.97.94.57)` is entered into bin `01101111` = 7



Example

Similarly, as `"c"` hashes to 3

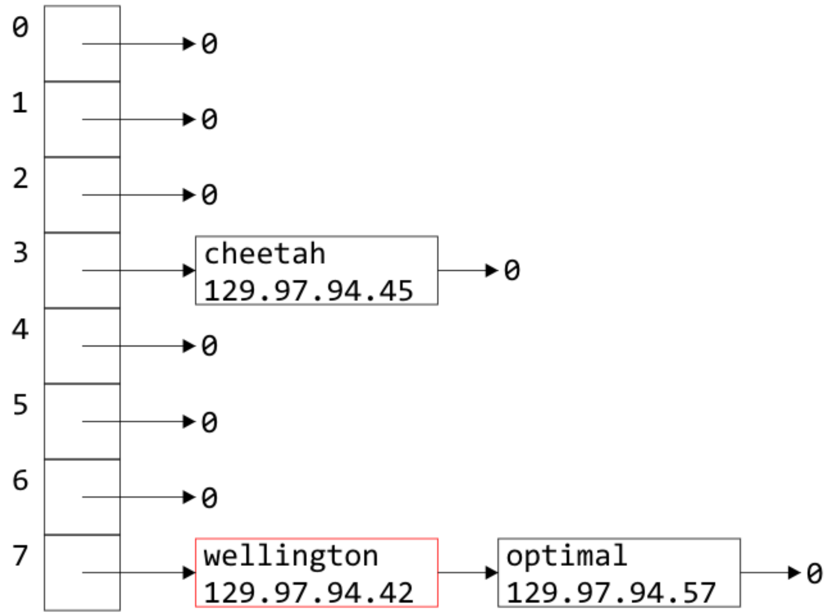
- The pair `("cheetah", 129.97.94.45)` is entered into bin 3



Example

The "w" in Wellington also hashes to 7

– ("wellington", 129.97.94.42) is entered into bin 7



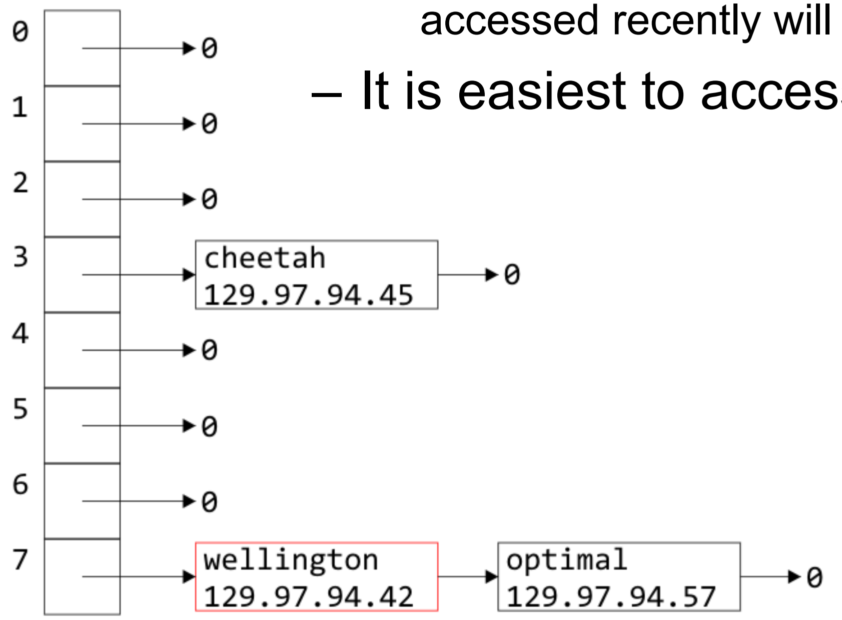
Example

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"

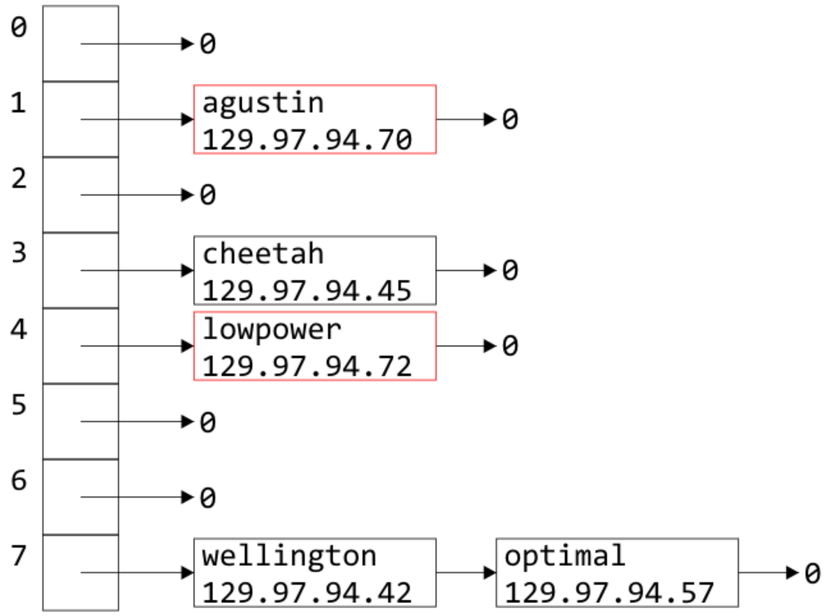
- It is easiest to access data at the front of a linked list



Heuristics include rules of thumb, educated guesses, and intuition

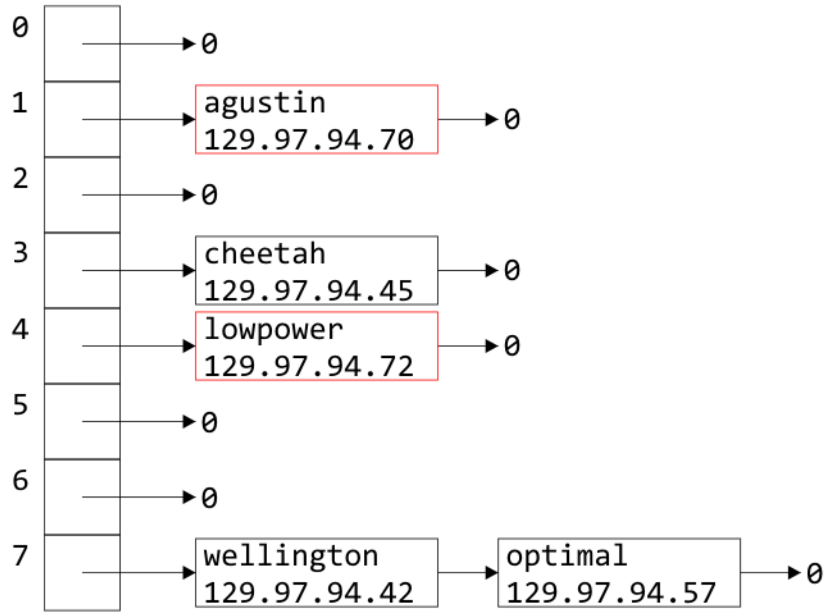
Example

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



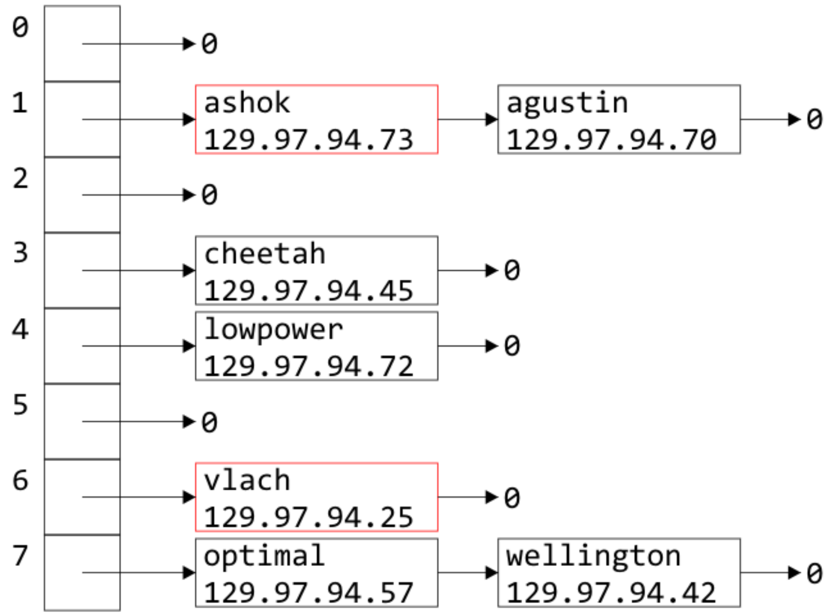
Example

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



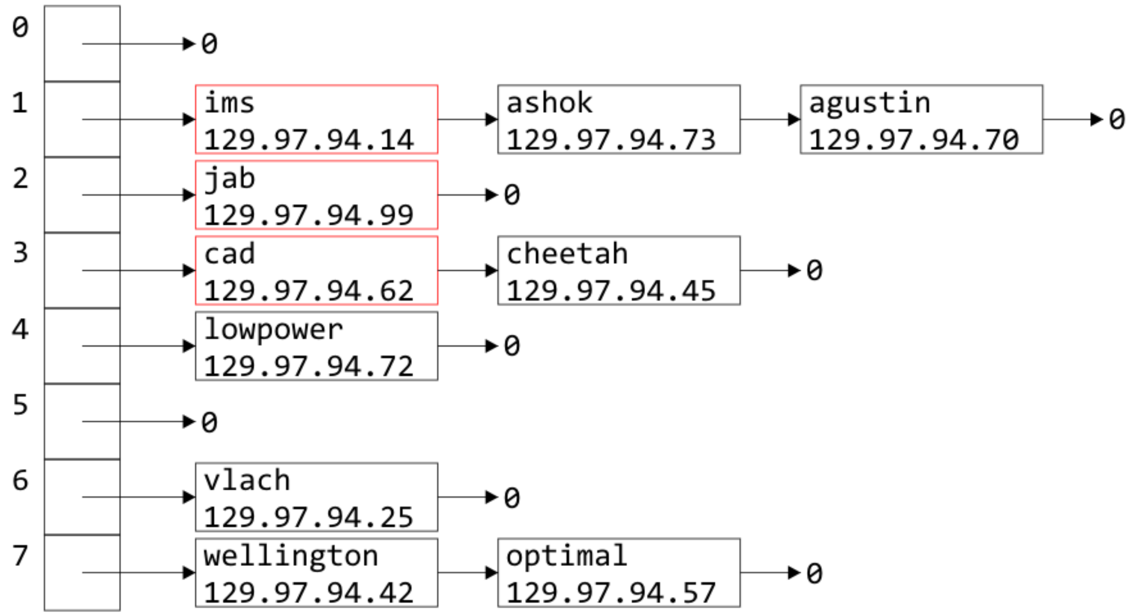
Example

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



Example

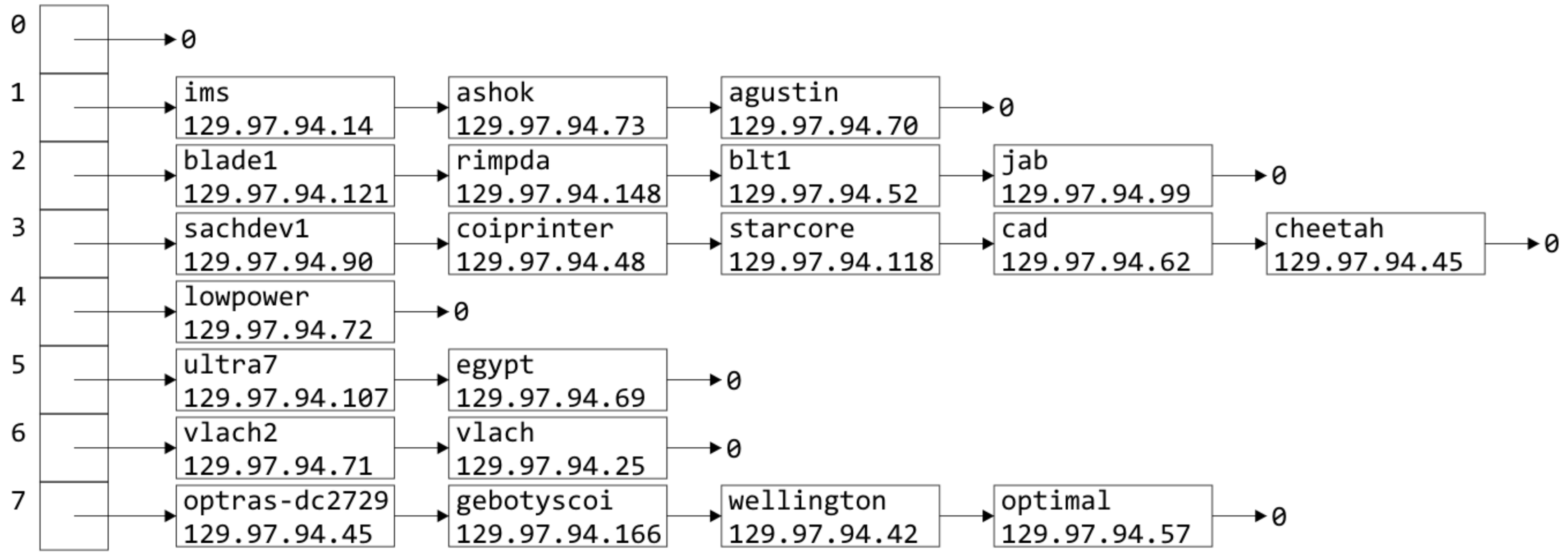
Inserting `"ims"`, `"jab"`, and `"cad"` doesn't even out the bins



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:

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

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

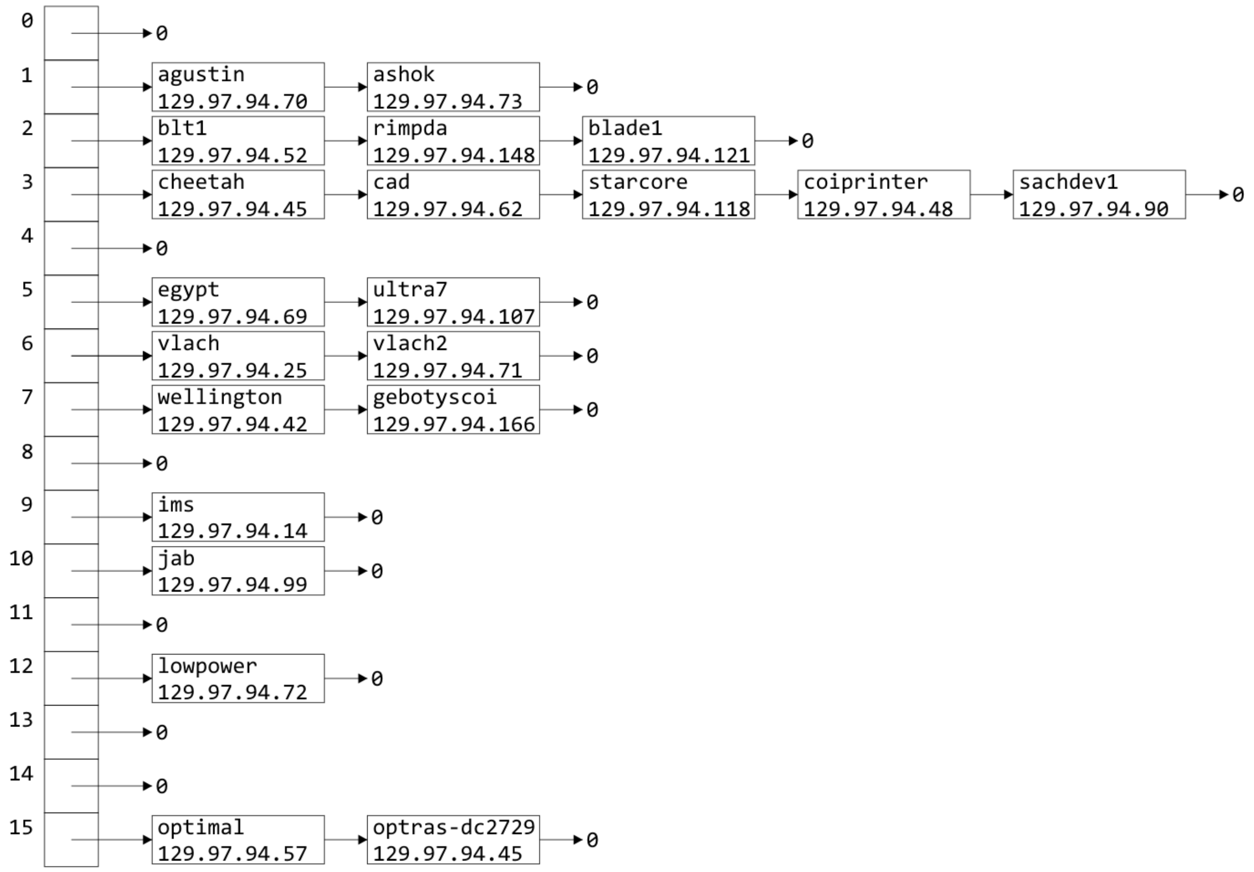
The most obvious solution is to double the size of the hash table

- 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 $\lambda = 1.3125$

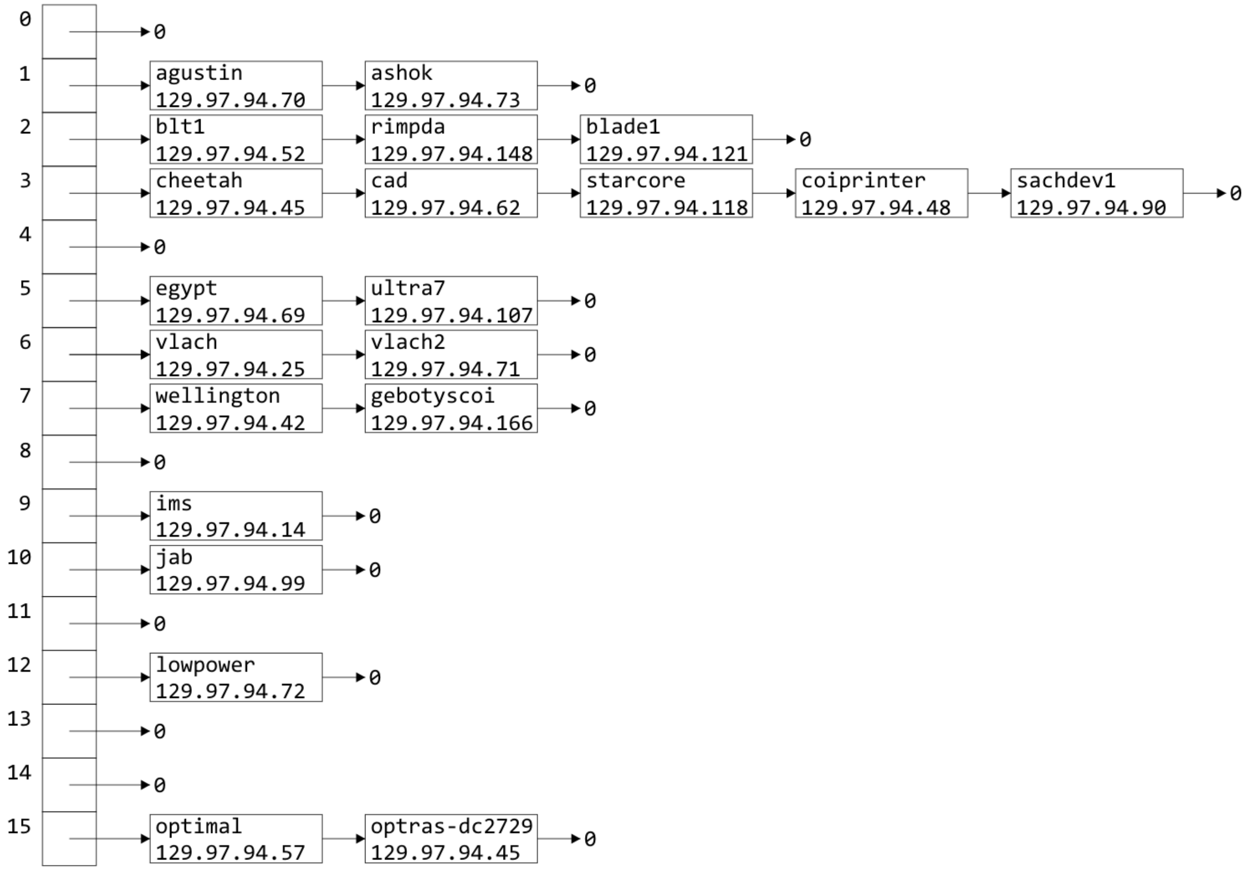
– Unfortunately, the distribution hasn't improved much



0 → 0
1 → agustin 129.97.94.70 → ashok 129.97.94.73 → 0
2 → blt1 129.97.94.52 → rimpda 129.97.94.148 → blade1 129.97.94.121 → 0
3 → cheetah 129.97.94.45 → cad 129.97.94.62 → starcore 129.97.94.118 → coiprinter 129.97.94.48 → sachdev1 129.97.94.90 → 0
4 → 0
5 → egypt 129.97.94.69 → ultra7 129.97.94.107 → 0
6 → vlach 129.97.94.25 → vlach2 129.97.94.71 → 0
7 → wellington 129.97.94.42 → gebotyscoi 129.97.94.166 → 0
8 → 0
9 → ims 129.97.94.14 → 0
10 → jab 129.97.94.99 → 0
11 → 0
12 → lowpower 129.97.94.72 → 0
13 → 0
14 → 0
15 → optimal 129.97.94.57 → optras-dc2729 129.97.94.45 → 0

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

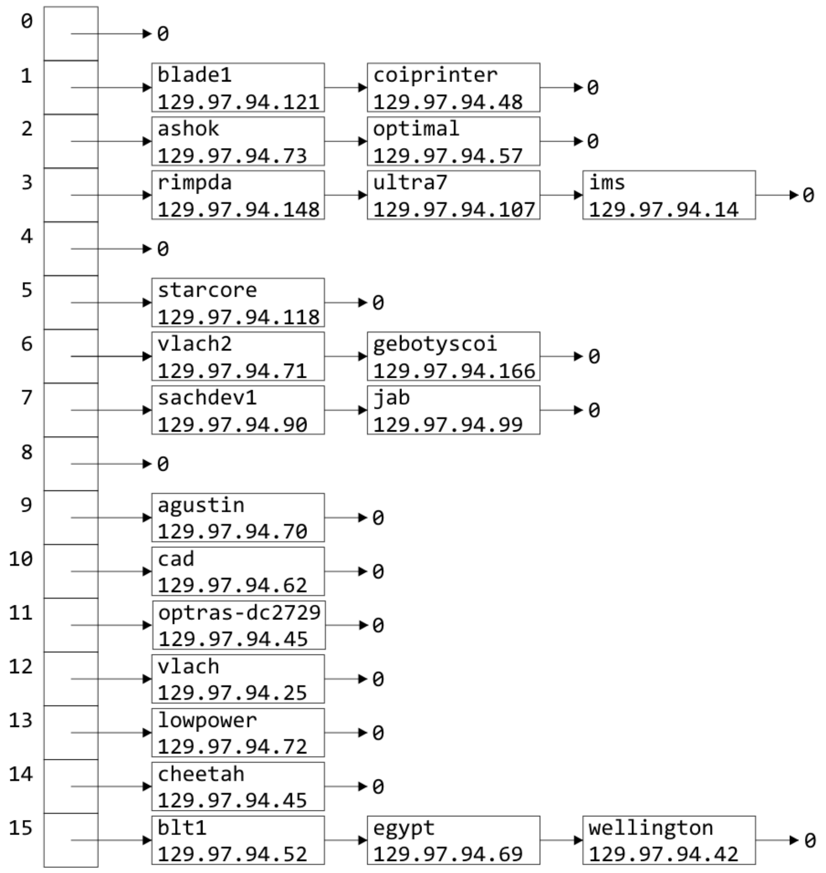
Choosing a Good Hash Function

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

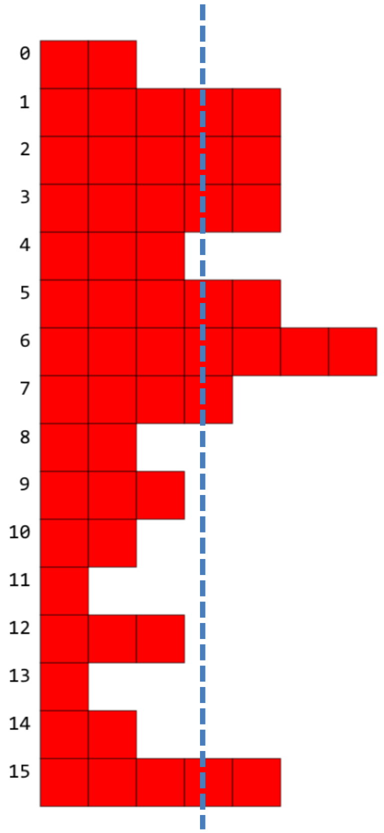
This hash function yields a much nicer distribution:



Choosing a Good Hash Function

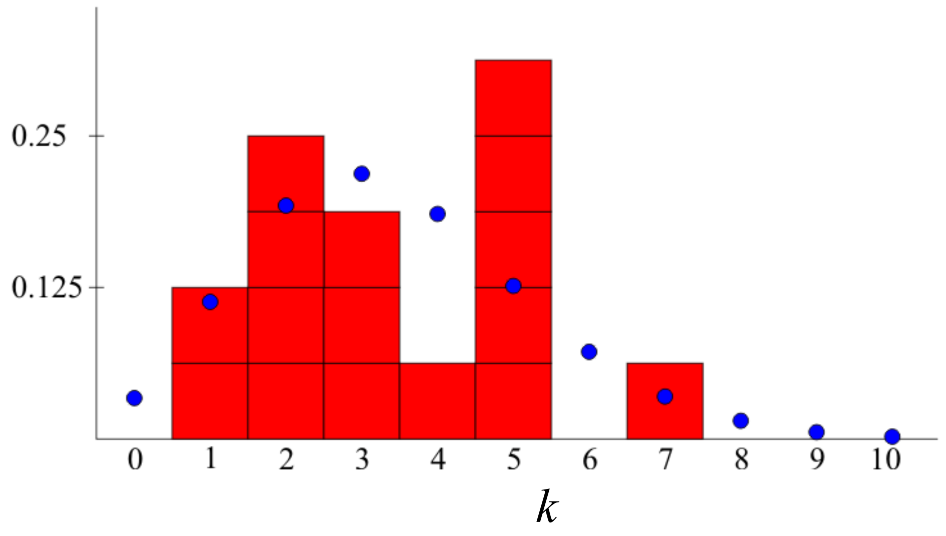
When we insert the names of all 55 names, we would have a load factor $\lambda = 3.4375$

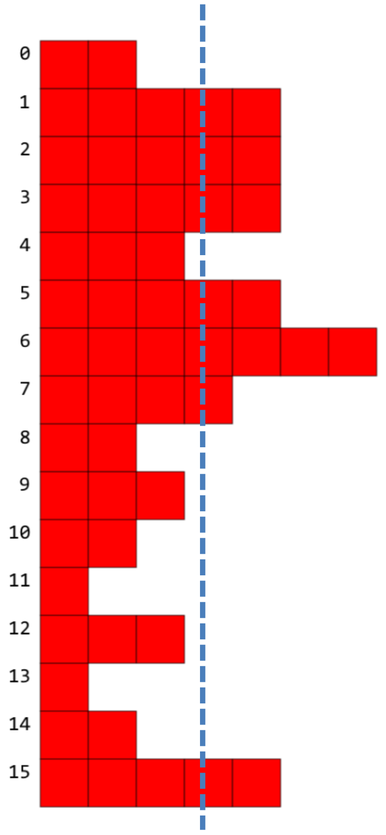
- Clearly there are not exactly 3.4375 objects per bin
- How can we tell if this is a good hash function?
- Can we expect exactly 3.4375 objects per bin?
  - Clearly no…
- The answer is with statistics…



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

References

Wikipedia, http://en.wikipedia.org/wiki/Hash_function

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

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