ECE430.217 Data Structures

Chained Hash Tables

Weiss Book Chapter 5

Byoungyoung Lee
https://compsec.snu.ac.kr
byoungyoung@snu.ac.kr

Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

Outline

We covered:

- Hash functions: how to map a key to an table index
- Hash functions may result in collisions

Now we must deal with collisions

- Numerous techniques exist

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

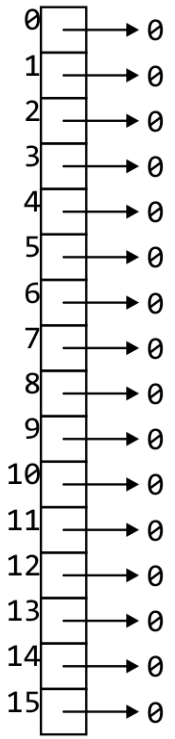
Consider associating each bin with a linked list:

```
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

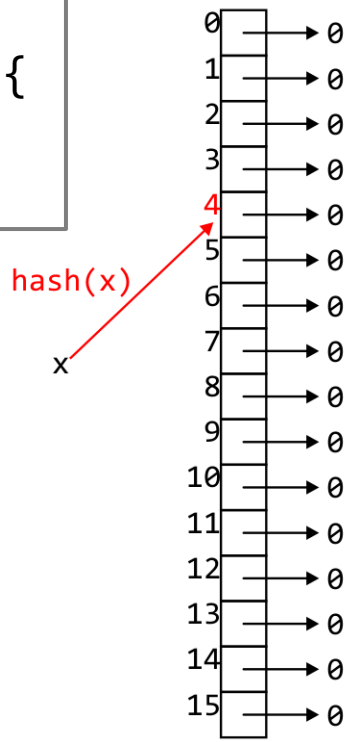
```
template <class Type>
Chained_hash_table::Chained_hash_table( int n ):
table_capacity( std::max( n, 1 ) ),
table_size( 0 ),
table( new Single_list<Type>[table_capacity] ) {
    // empty constructor
}
```



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() );
}
```



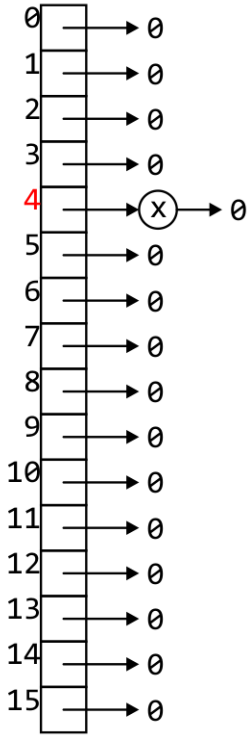
Recall:

- `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:

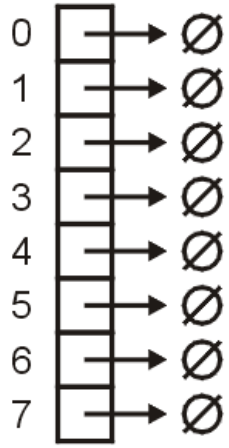
```
template <class Type>
void Chained_hash_table::insert( Type const &obj ) {
    unsigned int bin = hash( obj );

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



Example

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

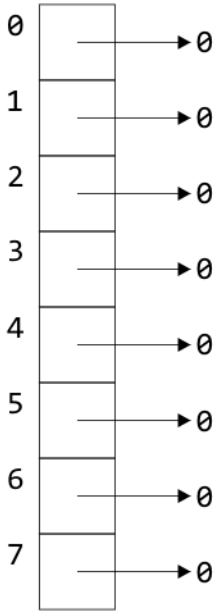
- We will choose the bin based on the host name
- Associated with the name will be the IP address
- *E.g.*, `("optimal", 129.97.94.57)`



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

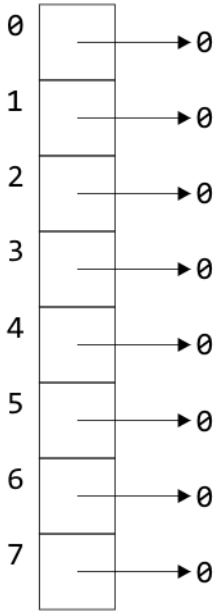
- The hash of `"optimal"` is based on `"o"`



Example

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

- "a" is 1 and it cycles from there…

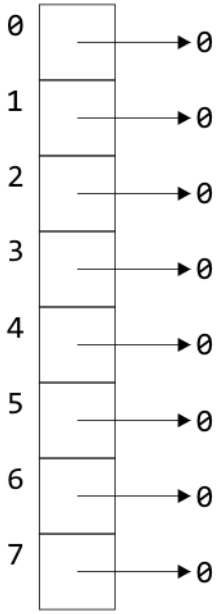


a	01100001	n	01101110
b	01100010	o	01101111
c	01100011	p	01110000
d	01100100	q	01110001
e	01100101	r	01110010
f	01100110	s	01110011
g	01100111	t	01110100
h	01101000	u	01110101
i	01101001	v	01110110
j	01101010	w	01110111
k	01101011	x	01111000
l	01101100	y	01111001
m	01101101	z	01111010

Example

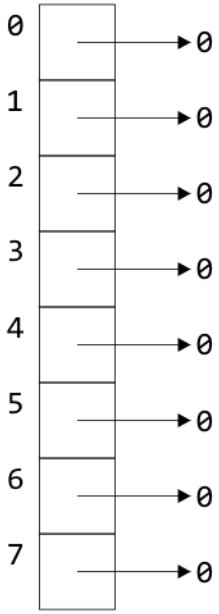
Our hash function is

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

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



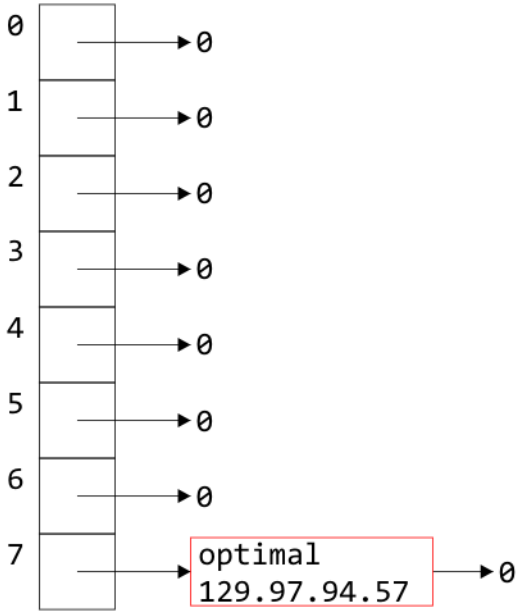
Example

Starting win an array of 8 empty linked lists



Example

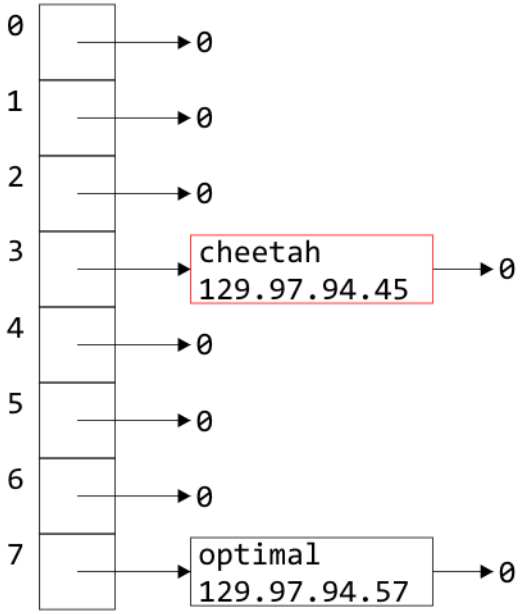
The pair `("optimal", 129.97.94.57)` is entered into bin `01101111` = 7



Example

Similarly, as `"c"` hashes to 3

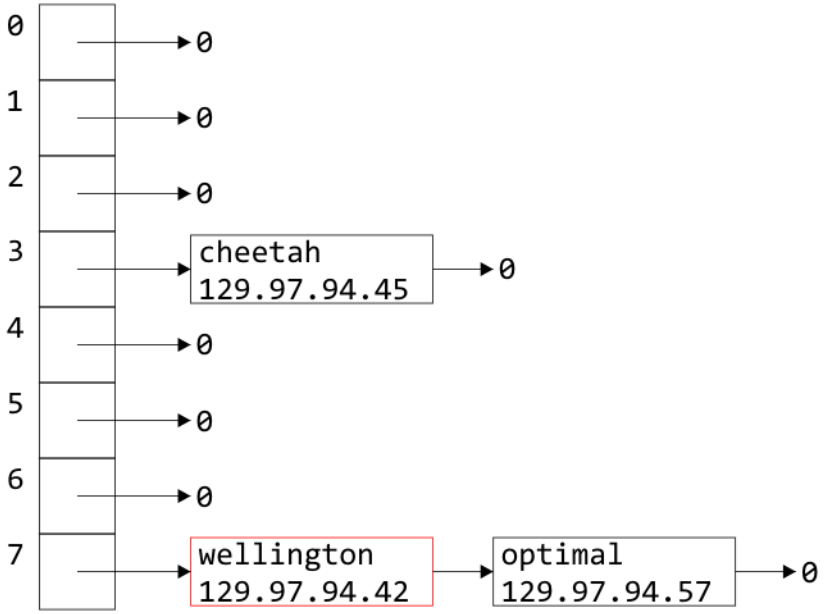
- The pair `("cheetah", 129.97.94.45)` is entered into bin 3



Example

The "w" in Wellington also hashes to 7

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

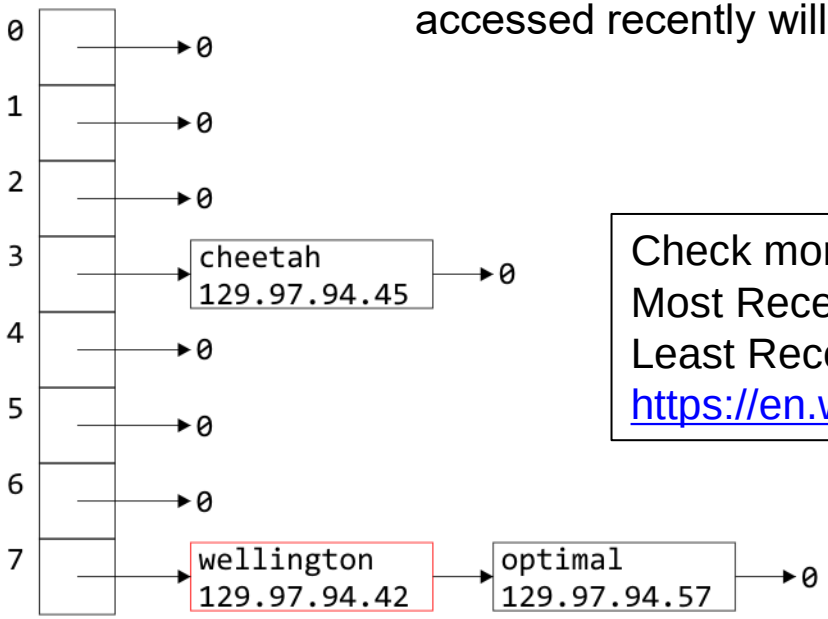


Example

Should we use `push_front` from the linked list?

- Do I have other choices?
- A good heuristic is

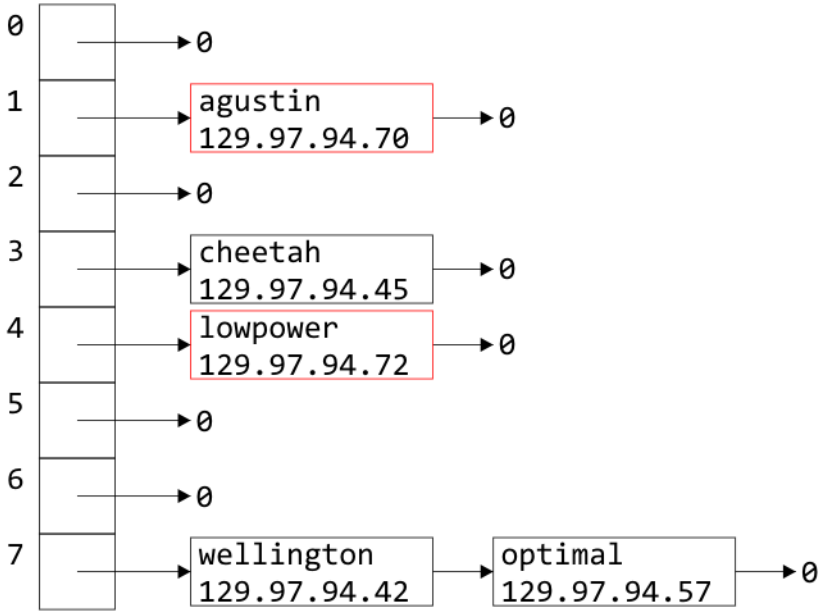
> "unless you know otherwise, data which has been accessed recently will be accessed again in the near future"



Check more:
Most Recently Used (MRU),
Least Recently Used (LRU)
https://en.wikipedia.org/wiki/Cache_replacement_policies

Example

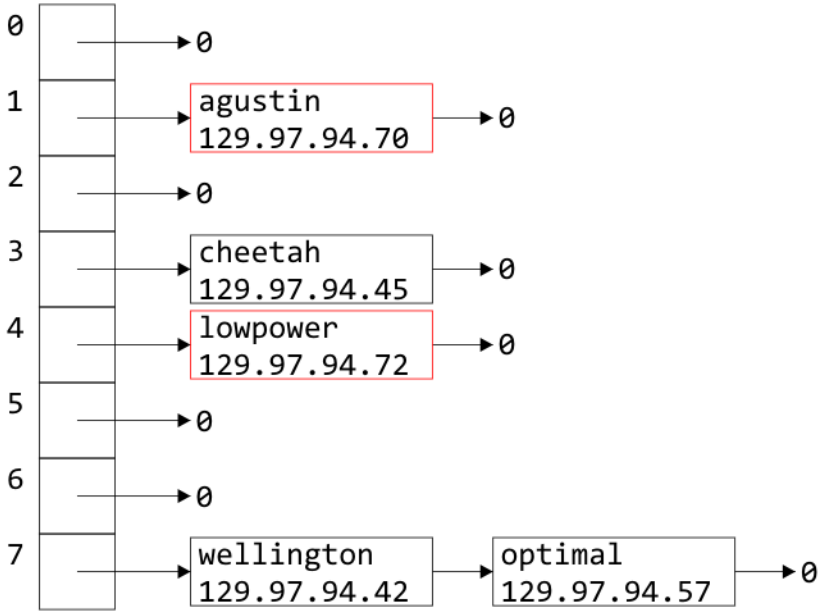
Similarly we can insert the host names `"augustin"` and `"lowpower"`



Example

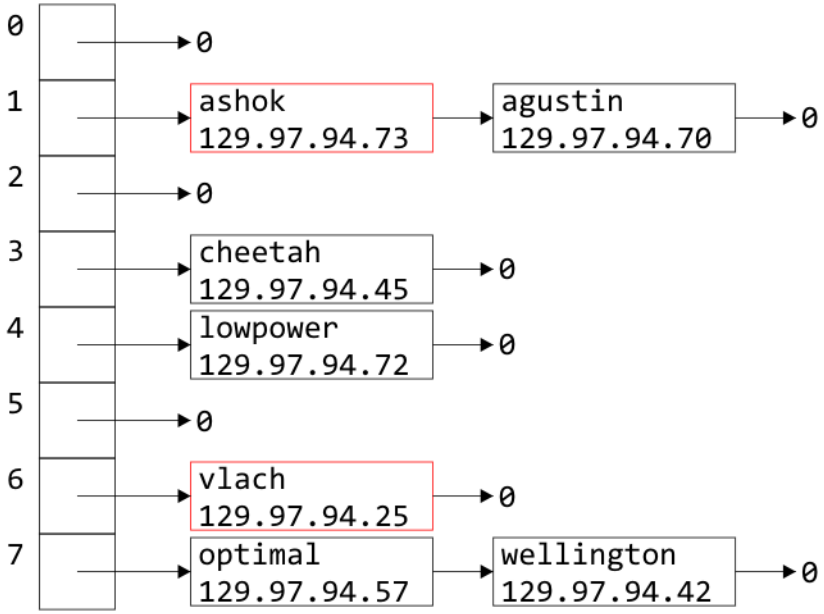
Search Operation: If we now wanted the IP address for `"optimal"`, we would

1) hash `"optimal"` to 7,
2) walk through the linked list, and
3) retrieve 129.97.94.57 from the corresponding node



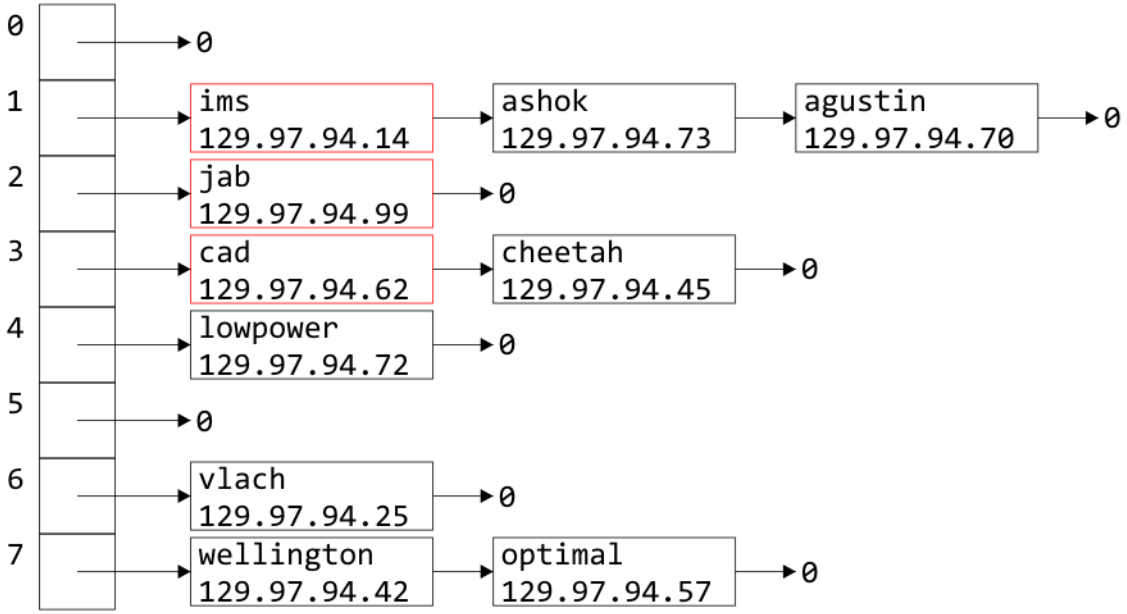
Example

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



Example

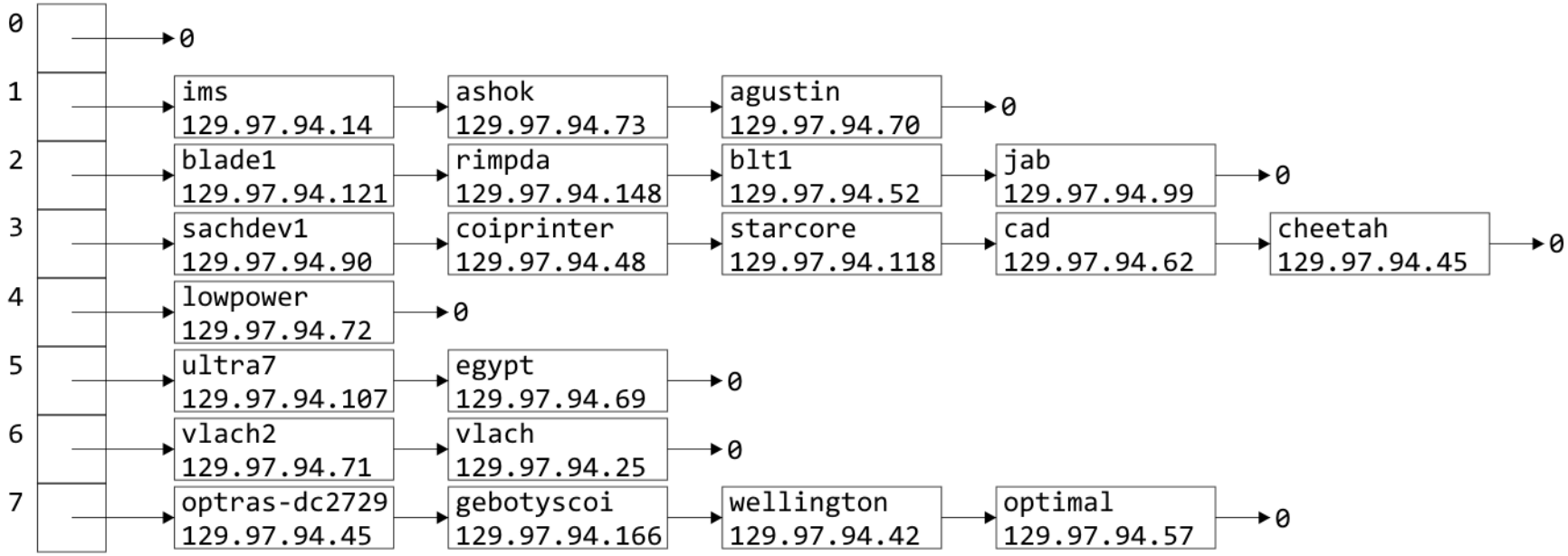
Inserting "ims", "jab", and "cad" doesn't even out the bins



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

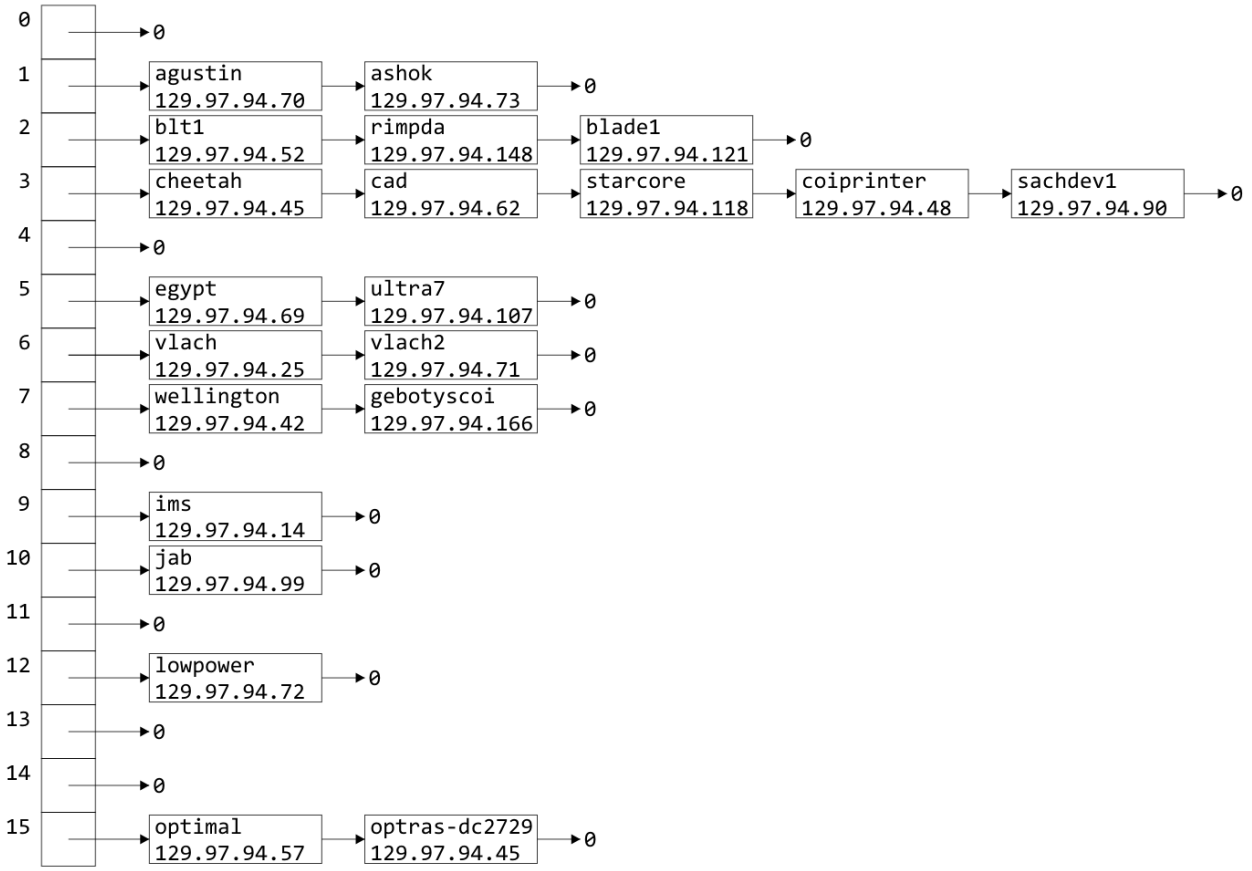
The most obvious solution is to double the size of the hash table

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

Doubling Size

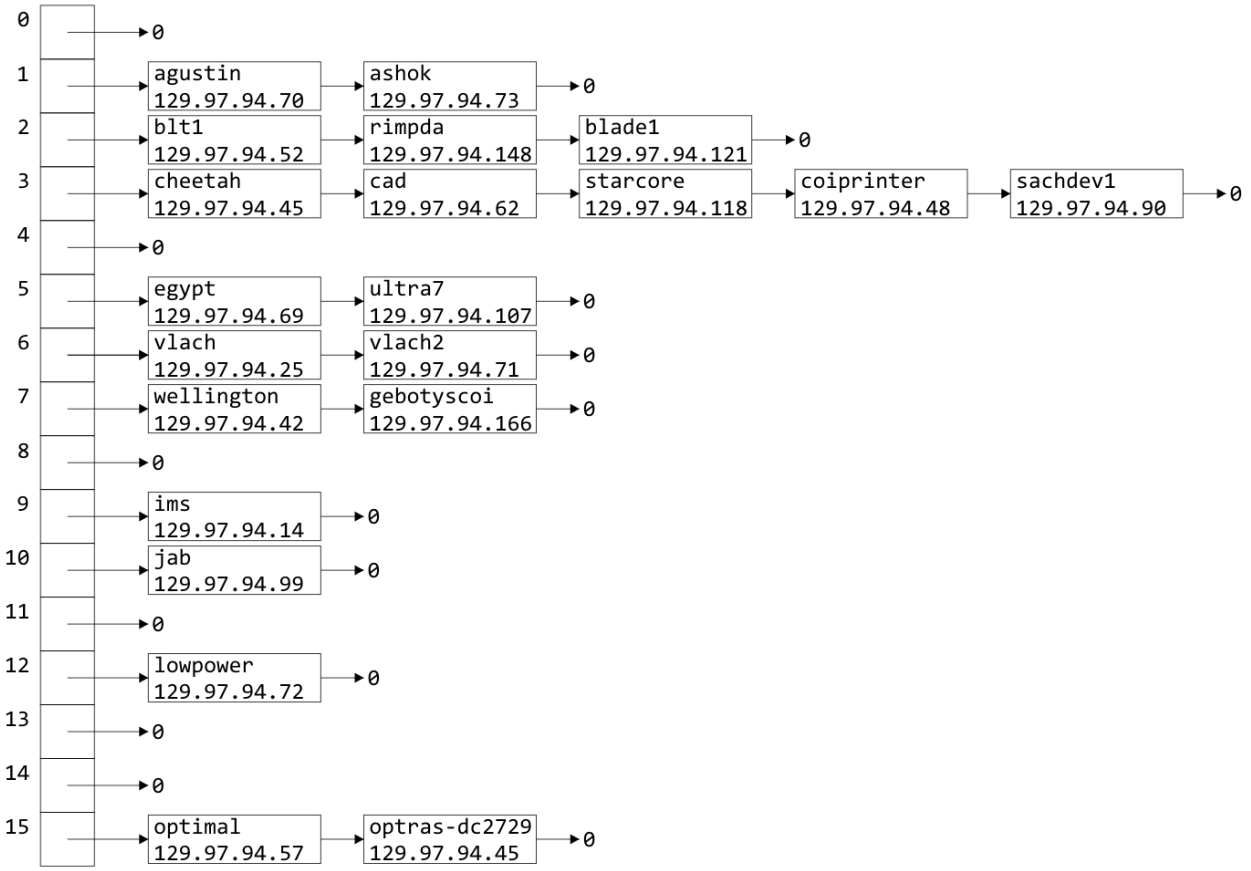
The load factor is now $\lambda = 1.3125$

- 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 chose a very poor hash function:

- We looked at the first letter of the host name

Suppose 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

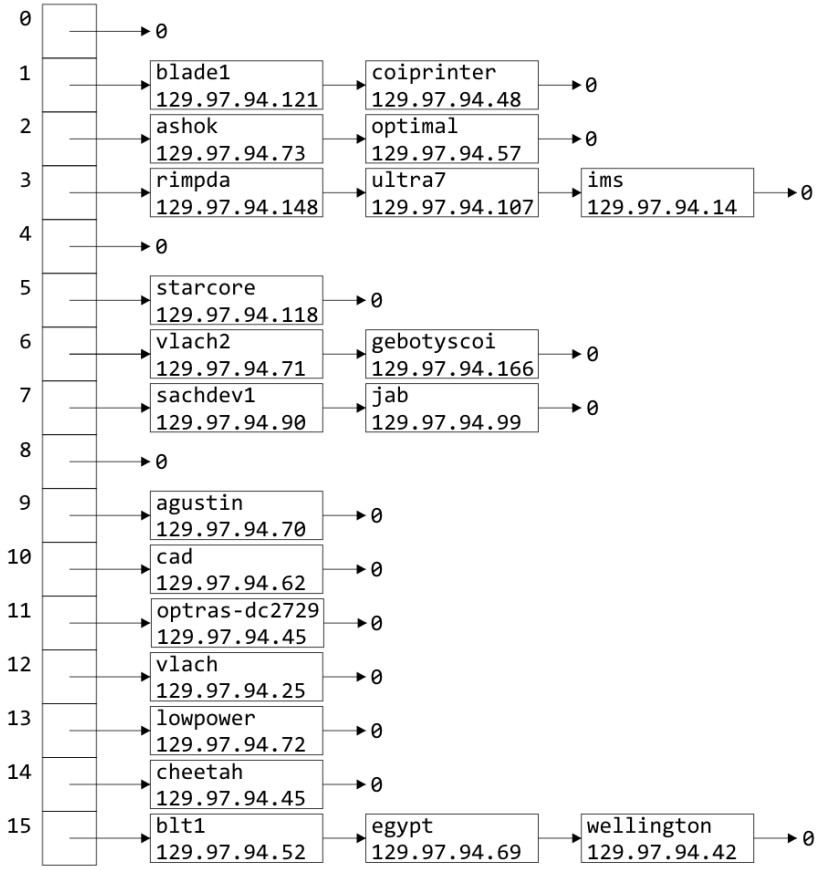
Choosing a Good Hash Function

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

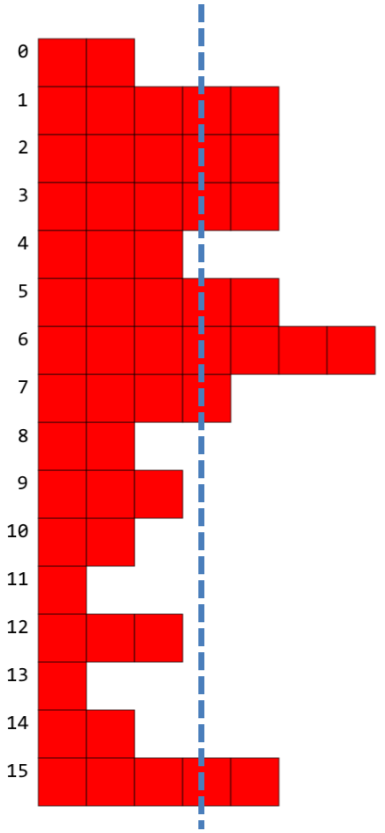
This hash function yields a much nicer distribution:



Choosing a Good Hash Function

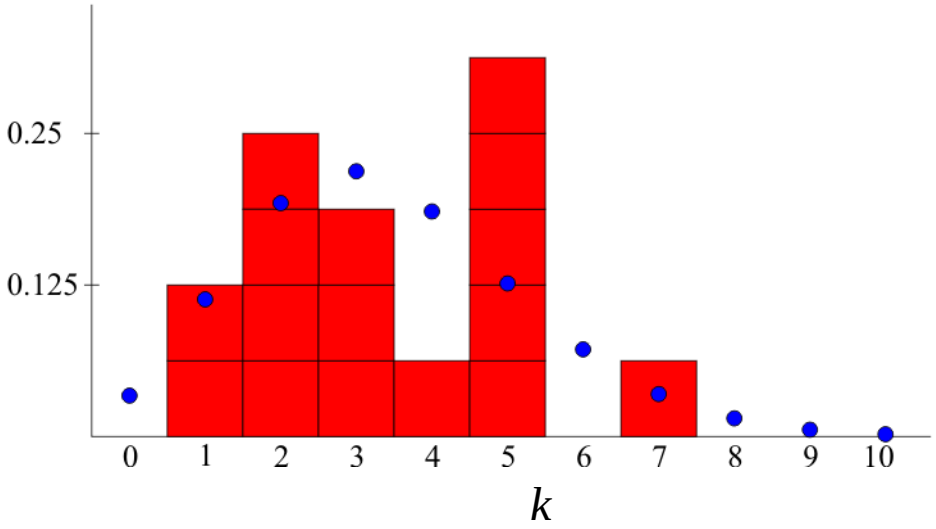
When we insert the names of all 55 names, we would have a load factor $\lambda = 3.4375$

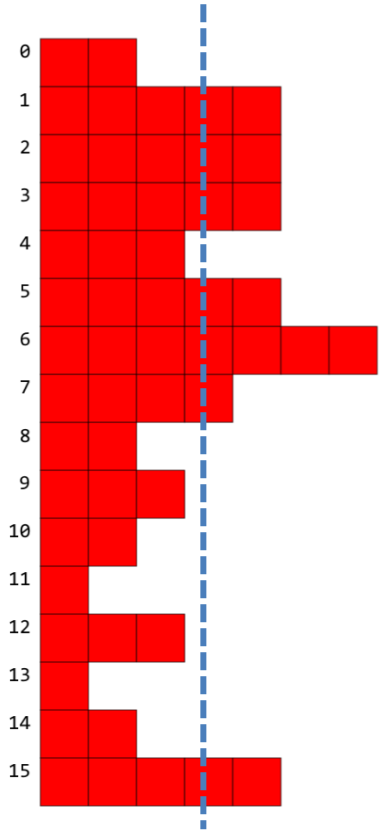
- Clearly there are not exactly 3.4375 objects per bin
- How can we tell if this is a good hash function?
- Can we expect exactly 3.4375 objects per bin?
  - Clearly no…
- The answer is with statistics…



Choosing a Good Hash Function

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





Poisson Distribution

- Suppose the average number of goals per match in a World Cup is 2.5 goals

- Q. What's the probability of having k goals in a match?
  - $P(k \text{ goals in a match}) = \frac{2.5^k e^{2.5}}{k!}$
  - The probability having 2 or 3 goals would be the highest
  - As *k* grows, the probability gradually decreases

k	$P(k$ goals in a World Cup soccer match)
0	0.082
1	0.205
2	0.257
3	0.213
4	0.133
5	0.067
6	0.028
7	0.010

Reference: https://en.wikipedia.org/wiki/Poisson_distribution

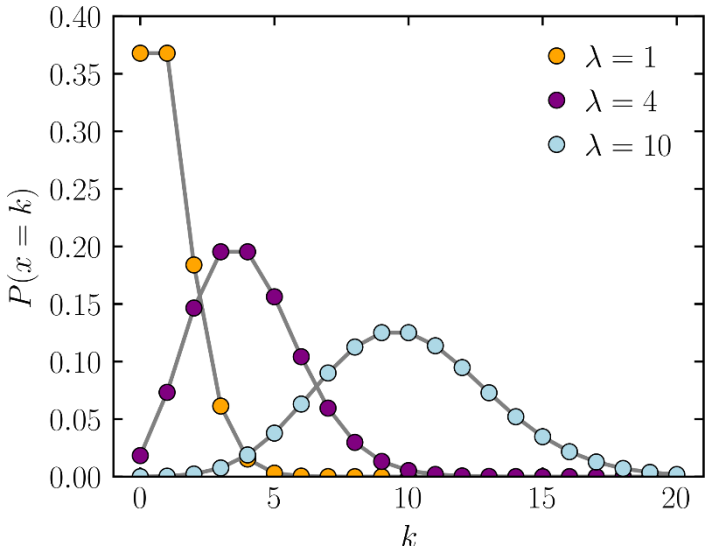
Poisson Distribution

- A discrete probability distribution
- The probability of a given number of events occurring in a fixed interval of time or space
  - Events occur with a known constant mean rate
  - Events occur independently since the last event
  - λ is the average number of events in a fixed interval

$$f(k;\lambda) = \Pr(X{=}k) = \frac{\lambda^k e^{-\lambda}}{k!},$$

where

- k is the number of occurrences ($k = 0, 1, 2...$)
- e is Euler's number ($e = 2.71828...$)
- ! is the factorial function.



Reference: https://en.wikipedia.org/wiki/Poisson_distribution

Property with Load Factor

- **Theorem**
  - In a chained hash table, **an unsuccessful search** takes average-case time $\Theta(1+\lambda)$, under the assumption of simple uniform hashing

- **Proof sketch**
  - A simple uniform hashing equally likely maps to each bin of M
  - Each bin in M has a linked list with λ elements in average
  - An unsuccessful search should walk through all linked list elements in a bin
  - $\Theta\left(1+\frac{\lambda}{M}M\right)=\Theta(1+\lambda)$
    - 1 indicates the time to compute the hash function

Property with Load Factor

- **Theorem**
  - In a chained hash table, **a successful search** takes average-case time $\Theta(1+\lambda)$, under the assumption of simple uniform hashing
- **Proof**
  - The successful search should take less than (or the same as) the unsuccessful search
  - Check more in CLRS 11.2 (p259)

Property with Load Factor

- **Suppose**
  - The number of hash table bins (i.e., M) is proportional to the number of elements in the hash table (i.e., n)
- **Consequently**
  - $n = O(M)$
  - $\lambda = \frac{n}{M} = \frac{O(M)}{M} = O(1)$
  - Thus, the searching takes $O(1)$ on average
- **Conclusion**
  - With an enough number of bins, the search of a hash table takes $O(1)$.

Problems with Linked Lists

One significant issue with chained hash tables using linked lists

- Requires extra memory
- Dynamic memory allocation
- Need to walk through a linked list: $\mathbf{O}(\lambda)$

Problems with Linked Lists

For faster access, we could **replace each linked list with a search 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

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.