An introduction to Hash Tables

Waqar Ahmad
Department of Computer Science
Syed Babar Ali School of Science and Engineering
Lahore University of Management Sciences (LUMS)

Outline

Discuss storing unordered data

- IP addresses and domain names

Introduce the idea of hashing:

- Reducing $\mathbf{O}(\log(n))$ 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 the conditions is identified by an 8-bit number (1 byte) 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 then call the appropriate function.

Supporting Example

```
#include <iostream>

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

void b() {
    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:

```
Calling 'void a()'
Calling 'void b()'
```

Supporting Example

- This is slow—we would have to do some form of search (e.g. binary search) in order to determine which of the 150 slots corresponds to, for example, the 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 indicates which bin (index) the corresponding function pointer was stored
  - For example, the pointer to the function corresponding to the error-code 15 would be at array index ‘15’.

In general, we would like to:

- Create an array of size M **(Hash table)**
- Store each of n objects in one of the M bins
- Have some means of determining the bin in which an object is stored (**Hash Function**)

IP Addresses

Examples:

Suppose we want to associate IP addresses and corresponding domain names

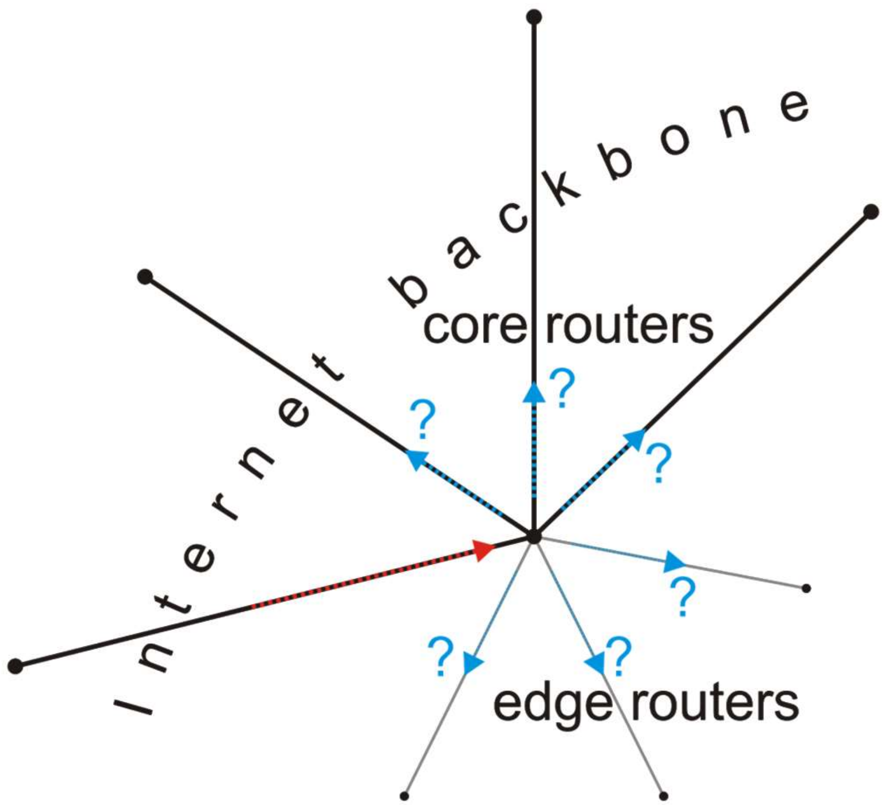
A 32-bit IP address is often written as four-byte values from 0 to 255

- 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

Consider core routers on the Internet:

- They must associate each IP address with the next router that address should be passed to using a routing table.



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 considering the class size of about a few hunderds?

Suppose each student is issued an 8-digit student id

- Suppose a student has the student id: 20253456
  - Expected year of graduation (4 digits on the left: 2025) and a serial number (4 digits on the right: 3456)
- 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

- Array indices would be from 0 to 999
- **How could you convert an 8-digit student id into a 3-digit number?**
- First four digits might cause a problem: almost all student ids start with 2025, 2024, 2023
- The last three digits, however, are essentially random

Therefore, I could store examination grade of a student with id 20253456 at index 456:

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

Simpler problem

Consequently, we can have a function that maps a student id onto a 3-digit number

- This 3-digit number is the array index where we can store data
- Storing it, accessing it, and erasing it is $\Theta(1)$
- Problem: two or more students may map to the same number:
  - Wang has Id 20253456 and scored 85
  - Ahmed has Id 2024456 and scored 87

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

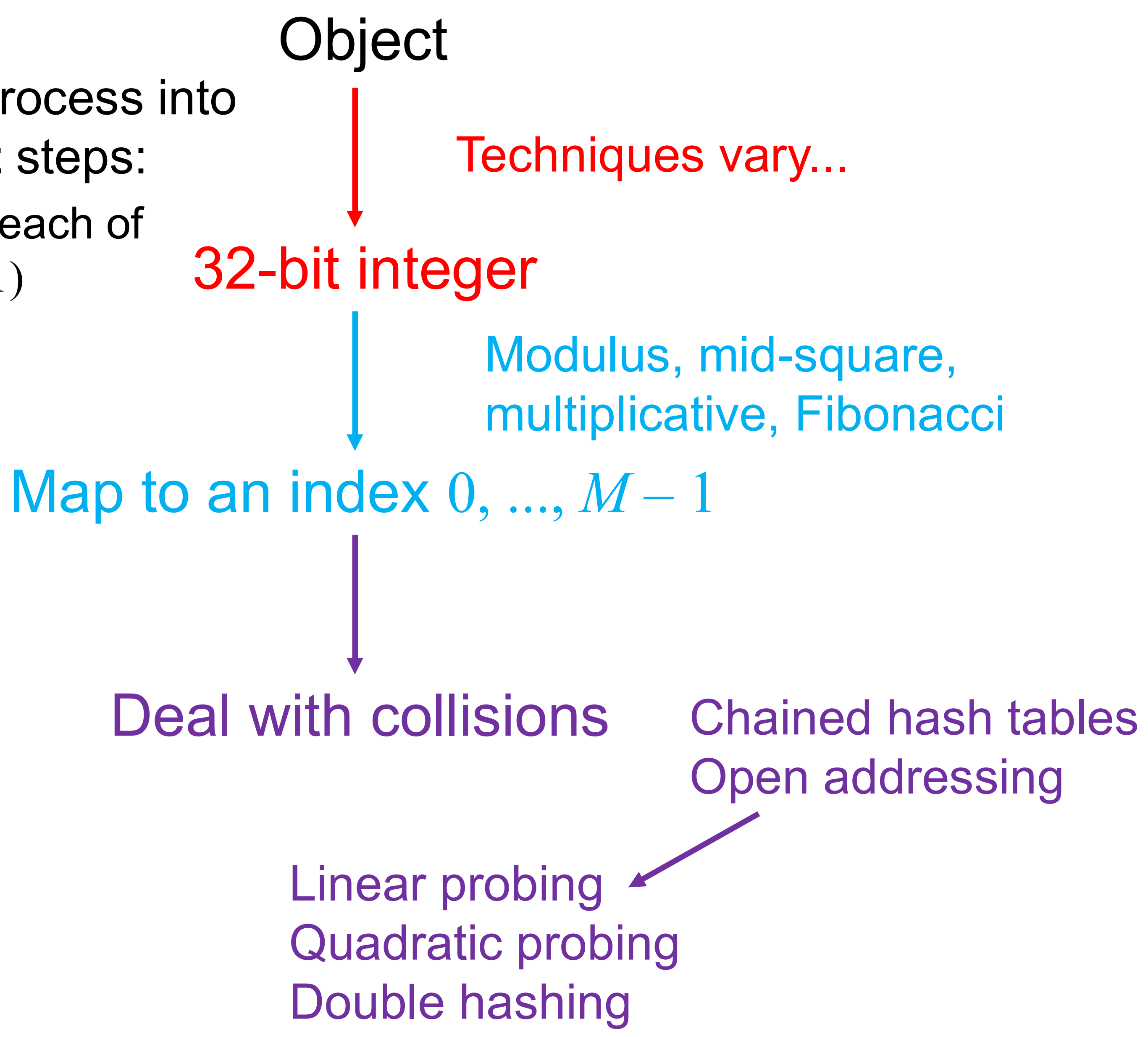
The hashing problem

- The process of mapping an object or a number onto an integer in a given range is called *hashing*
  - For example, map an 8-digit student id to a 3-digit array index
- 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 hashing process

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

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



Definitions

What is 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`

Definitions

- A hash table has the following components:
  - An array of size M (called a table)
  - A mathematical function – called a hash function – that maps keys to valid array indices
    - hash_function: key → 0 .. M – 1

- Table entries are stored and retrieved by applying the hash function to the key to get the index used to probe the table

A hash function is a mathematical function that takes an input (key) and maps it to an index within the array (0 to M - 1).
The purpose of the hash function is to take the key (like a student ID, name, or other identifier) and convert it into a valid index in the table.
This ensures that each key has a specific place in the array, allowing data to be stored and retrieved quickly.
How it Works
Storing Data: When you want to store a value in the hash table, you pass its key through the hash function to get an index. This index tells you exactly where in the array to place the value.
Retrieving Data: To retrieve the value, you again pass the key through the hash function to get the index, and you use that index to access the stored value in constant time, (1).

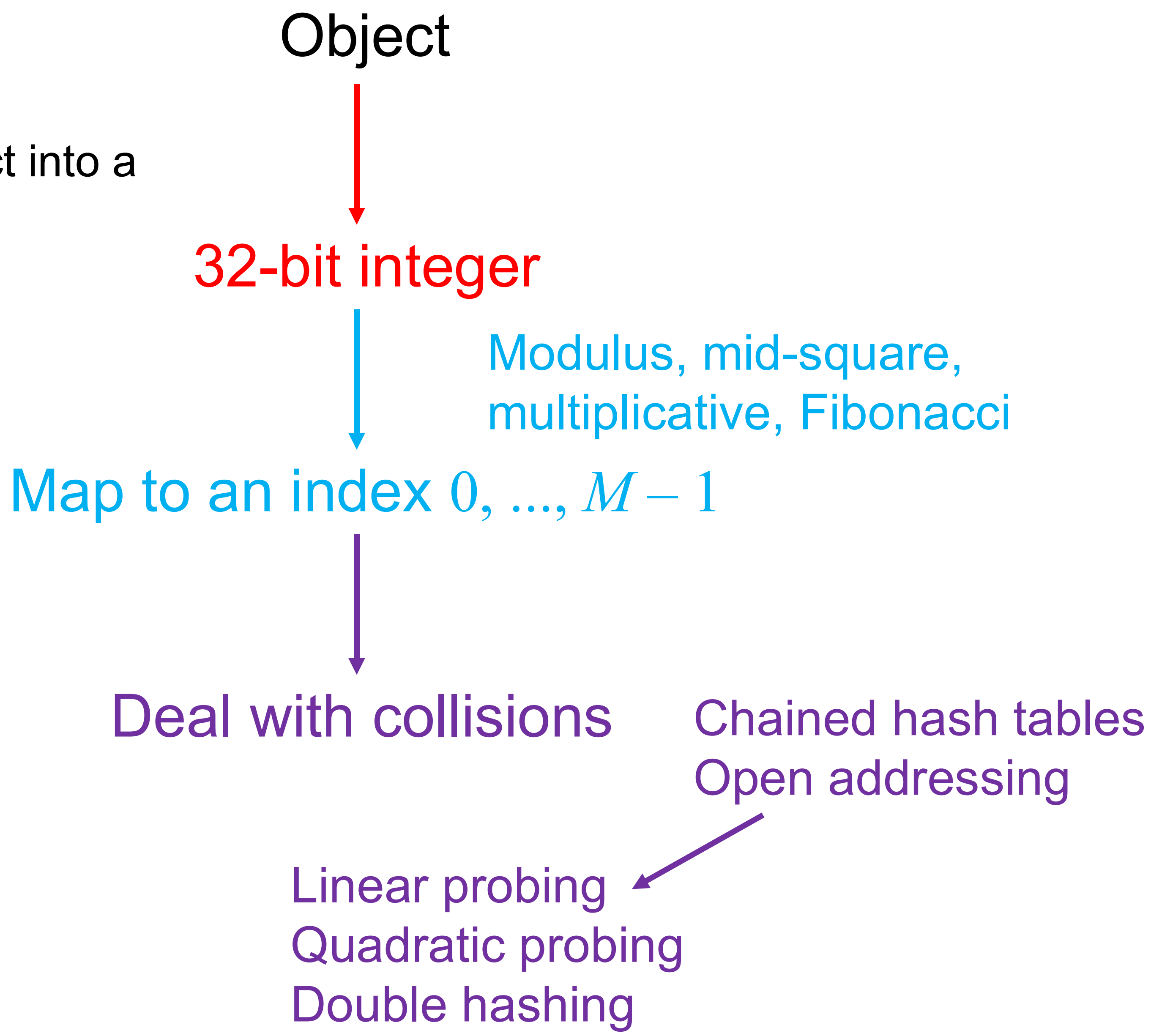
Suppose you have a hash table of size M = 10, and a key (like a student ID) is "2023". You apply the hash function to "2023" to get an index (say, index = 5). You then store the data at table[5]. To retrieve it later, you pass "2023" through the hash function again, get index = 5, and access table[5] to get the data.

Purpose of a Hash Table
Hash tables are used when you need fast lookups and have a limited number of keys, making them ideal for scenarios like dictionaries, databases, or any structure where you frequently store and access data by unique keys.

The hashing process

Let's start with:

– Hashing an object into a 32-bit integer



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

Collision occurs when two different objects are mapped to the same index in the table. Collisions are inevitable in hash tables due to the limited number of slots in the array.

There are several common techniques to handle collisions:

Chained Hash Tables: Each slot in the hash table can store a list of items. If multiple items hash to the same slot, they are stored in a linked list (or chain) within that slot.
Open Addressing: If a slot is already taken, the algorithm searches for another slot to place the item. Different strategies are used in open addressing:
Linear Probing: If a collision occurs, it checks the next slot (i.e., index + 1) until it finds an empty slot.
Quadratic Probing: Checks slots in a quadratic sequence (e.g., index + 1, index + 4, index + 9, etc.).
Double Hashing: Uses a second hash function to calculate an offset, which is added to the current index to find the next slot.

Object 32-bit integer (via hash function) Index in the table (0 to M-1).

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 LUMS students have a hash value:

– Student ID Number

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

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

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!";
```

Arithmetic Hash Values

An arithmetic hash value is calculated using a deterministic function that computes hash value based on relevant member variables of an object

are objects and key the same thing?

We will look at arithmetic hash functions for:

- Rational numbers, and
- Strings

Rational number class

What if we just add the numerator and denominator to get a hash value?

```
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 );
}
```

Do you expect any collisions?

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;
}
```

Not very good:

- Slow run time: $\Theta(n)$: n is the length of the string
- Words with the same characters hash to the same code:
  - `"form"` and `"from"`

The hashing process

Next we:

Map the key down to $0, \ldots, M-1$

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

Mapping down to $0, ..., M-1$

Previously, we considered calculation of 32-bit hash values of objects

- Explicitly defined hash values
- Implicitly calculated hash values using a function

Practically, we will require a hash value on the range $\mathbf{0, ..., \mathit{M}-1}$:

- **The modulus operator %**
- **The mid-squared**

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 $M - 1$

Common Hashing Functions: Modulus operator

Return the value modulus M

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

M is hash table size

Common Hashing Functions: Mid-Square

- The value is squared and the middle part of the result is taken as hash value.
- For example, to map the key **`3121`** into a hash table of size **`1000`**, we square it $3121^2 = 9740641$ and extract **`406`** as the hash value.
- Works well if the keys do not contain a lot of leading or trailing zeros.

The hash process

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

Next:
We must deal with collisions

Outline

We have:

- Discussed techniques for hashing objects to 32-bit integers
- Discussed mapping 32-bit integers down to a given range $0, ..., M - 1$

Now we must deal with collisions

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

Background

- 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 need to compe 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

The object is the actual data or item you want to store in the hash table.
The key is a unique identifier or attribute of the object that is used to calculate the hash value.
The value is what is stored in the hash table at the slot designated by the hash value of the key.

Implementation

Consider associating each bin with a linked list:

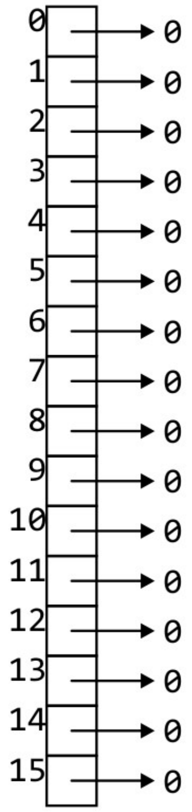
```
template <class Type>
class Chained_hash_table {
    private:
        int table_capacity;
        int table_size;
        Singly_list<Type> *table;

        unsigned int hash( Type const & );

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

The table array contains linked lists (Singly_list<Type>) that store elements with the same hash.

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 Singly_list<Type>[table_capacity] ) {
    // empty constructor
}
```



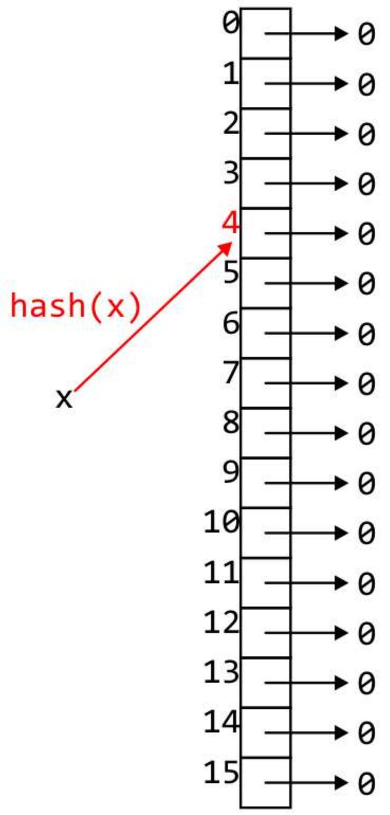
Implementation

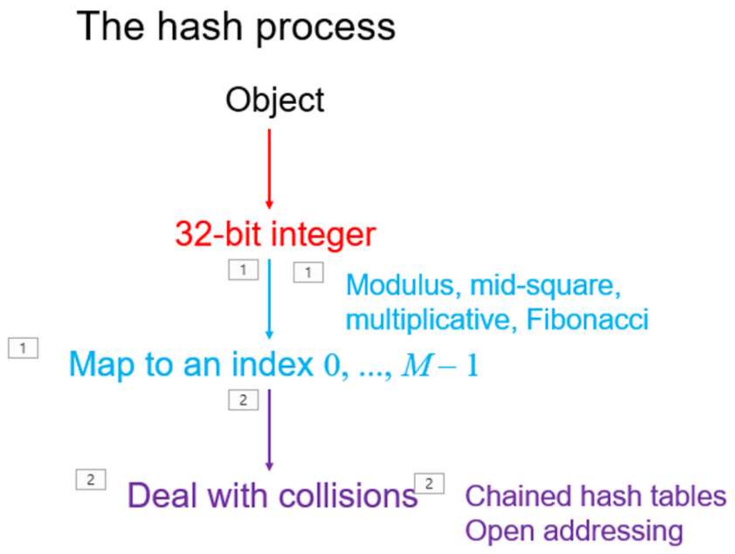
The goal is to determine the bin of an object:

```
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 $0, \ldots, M - 1$

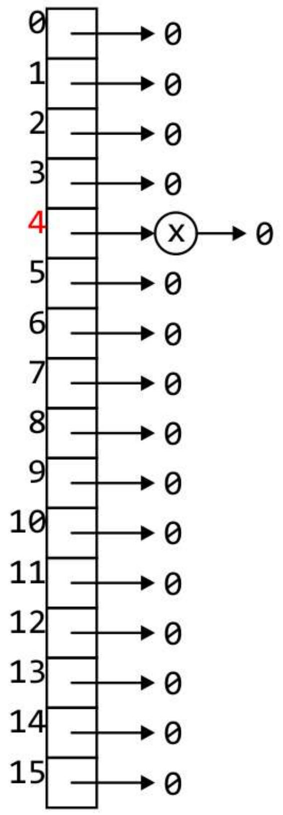




Implementation

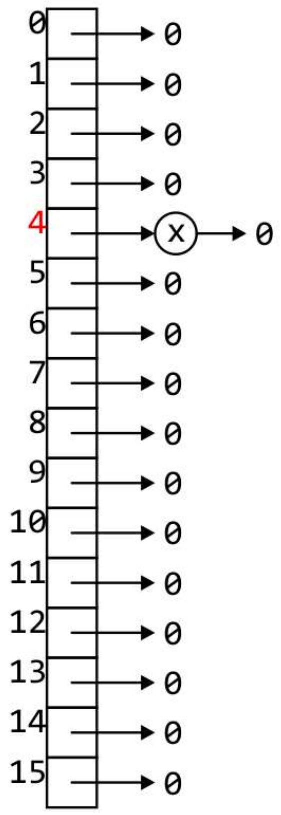
Other functions mapped to corresponding linked list functions:

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

        table[bin].push_front( obj );
        ++table_size;

}
```



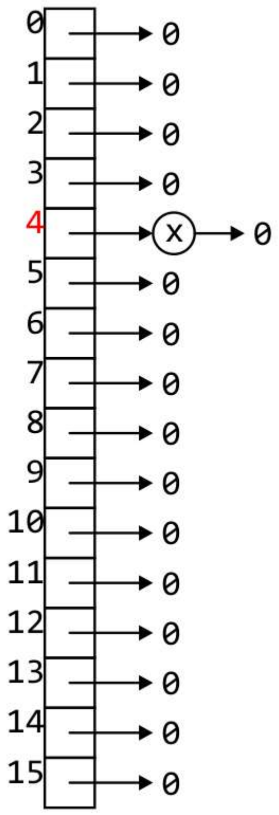
Implementation

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



Implementation

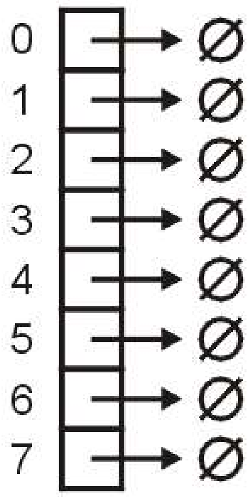
```
template <class Type>
int Chained_hash_table::erase( Type const &obj ) {
    unsigned int bin = hash( obj );

    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 addresses

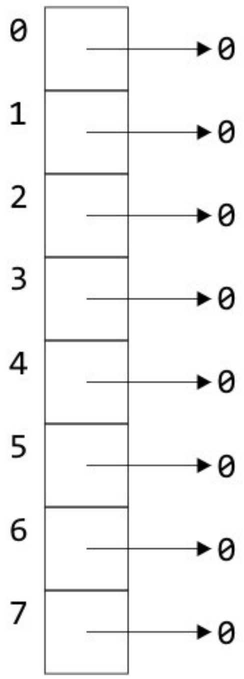
- We will choose the bin based on the host name
- Associated with the name will be the IP address
- *e.g.*, `("optimal", 129.97.94.57)`



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

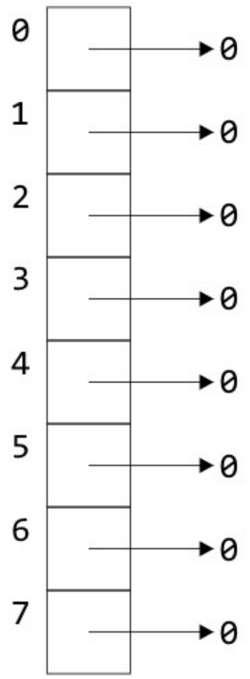
– The hash of "`optimal`" is based on "`o`"



Example

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

– "a" is 1 and it cycles from there…

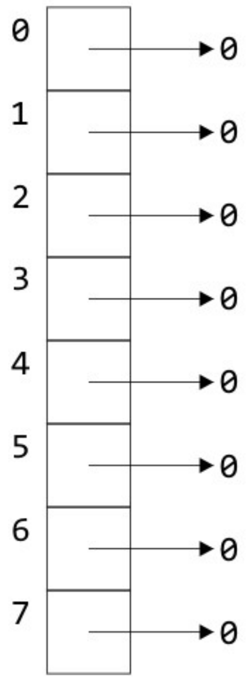


a	01100001	n	01101110
b	01100010	o	01101111
c	01100011	p	01110000
d	01100100	q	01110001
e	01100101	r	01110010
f	01100110	s	01110011
g	01100111	t	01110100
h	01101000	u	01110101
i	01101001	v	01110110
j	01101010	w	01110111
k	01101011	x	01111000
l	01101100	y	01111001
m	01101101	z	01111010

Example

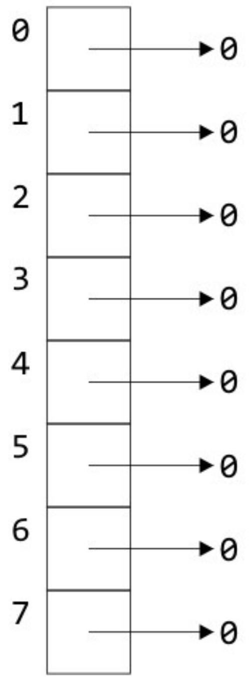
Our hash function is

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

     return str[0] % 8;
}
```



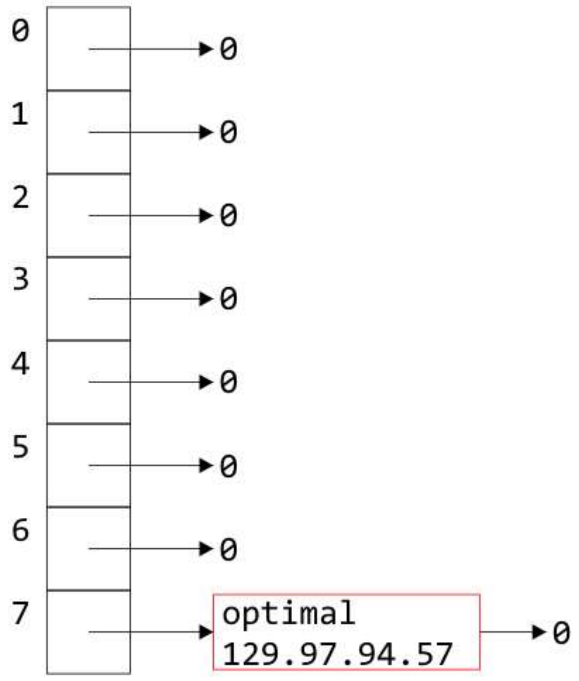
Example

Starting with an array of 8 empty linked lists



Example

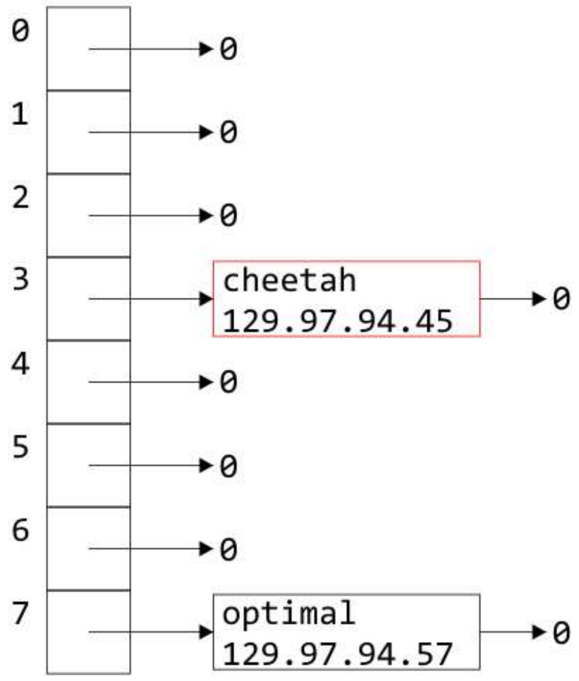
The pair (`"optimal", 129.97.94.57`) is entered into bin `01101111` = 7



Example

Similarly, as `"c"` hashes to 3

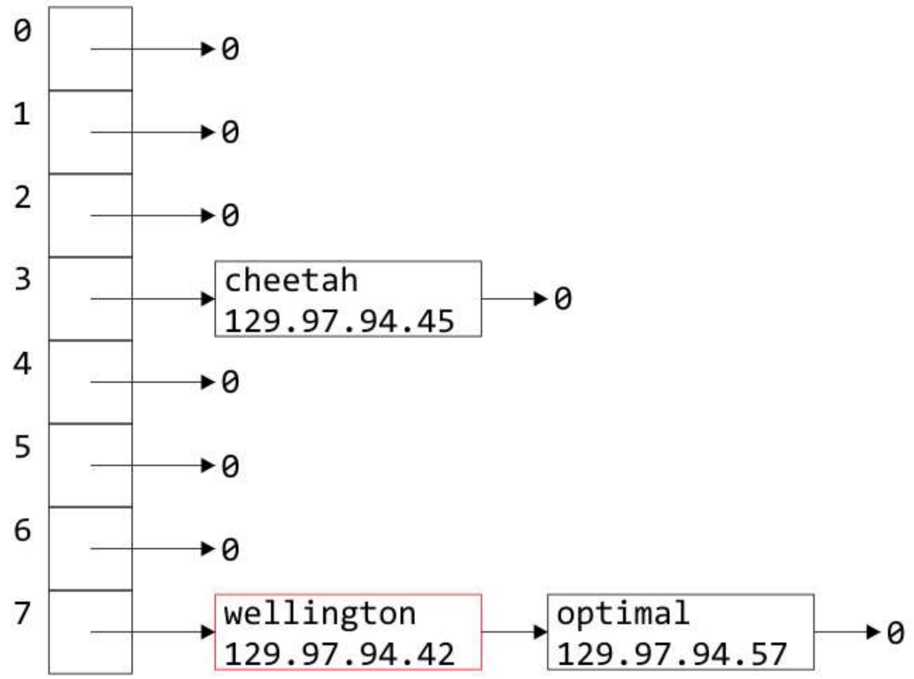
- The pair `("cheetah", 129.97.94.45)` is entered into bin 3



Example

The `"w"` in Wellington also hashes to 7

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



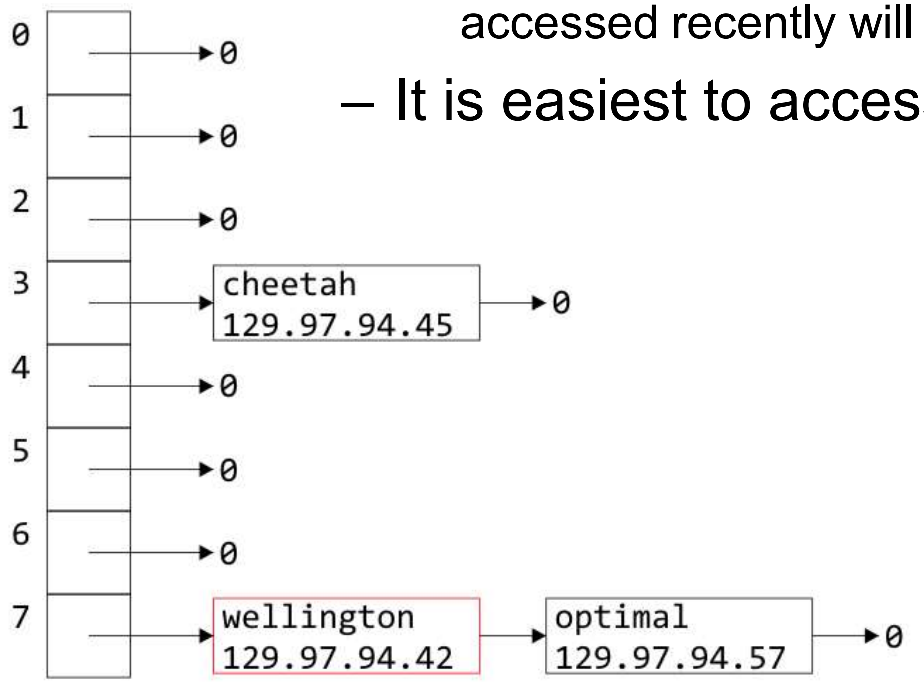
Example

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

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

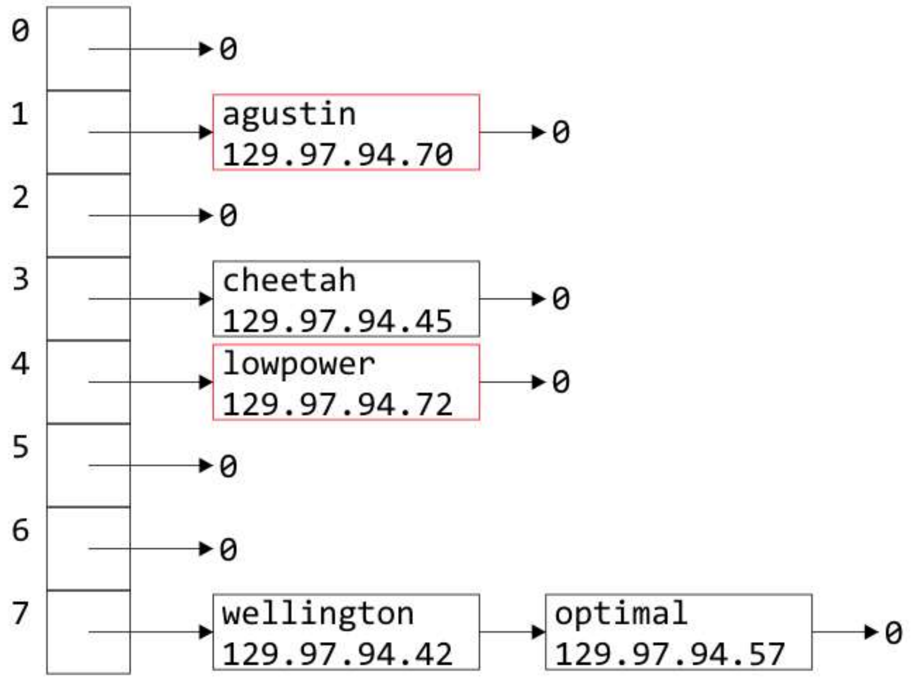
  "unless you know otherwise, data which has been accessed recently will be accessed again in the near future"

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



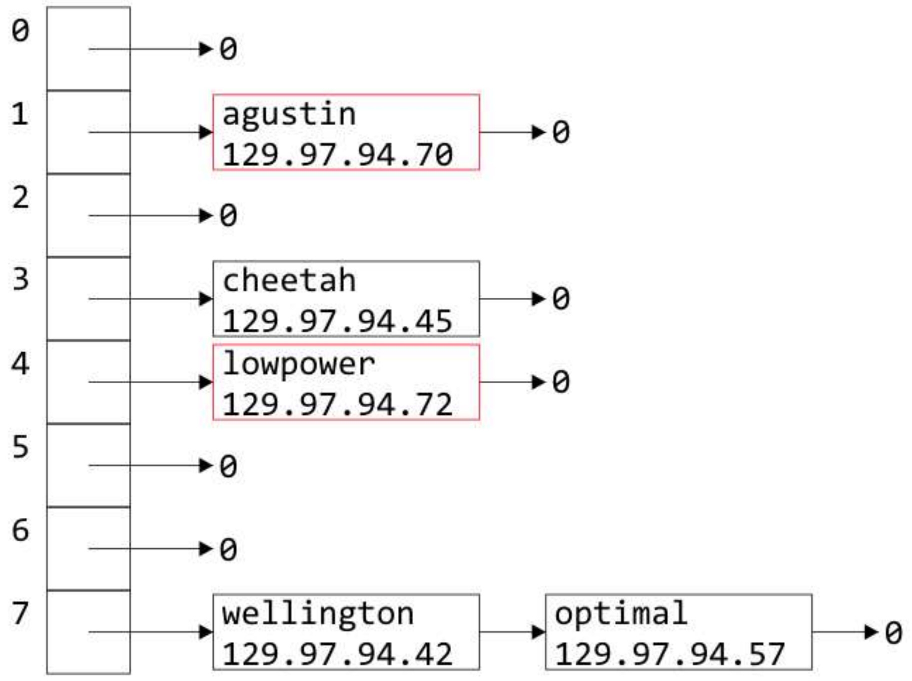
Example

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



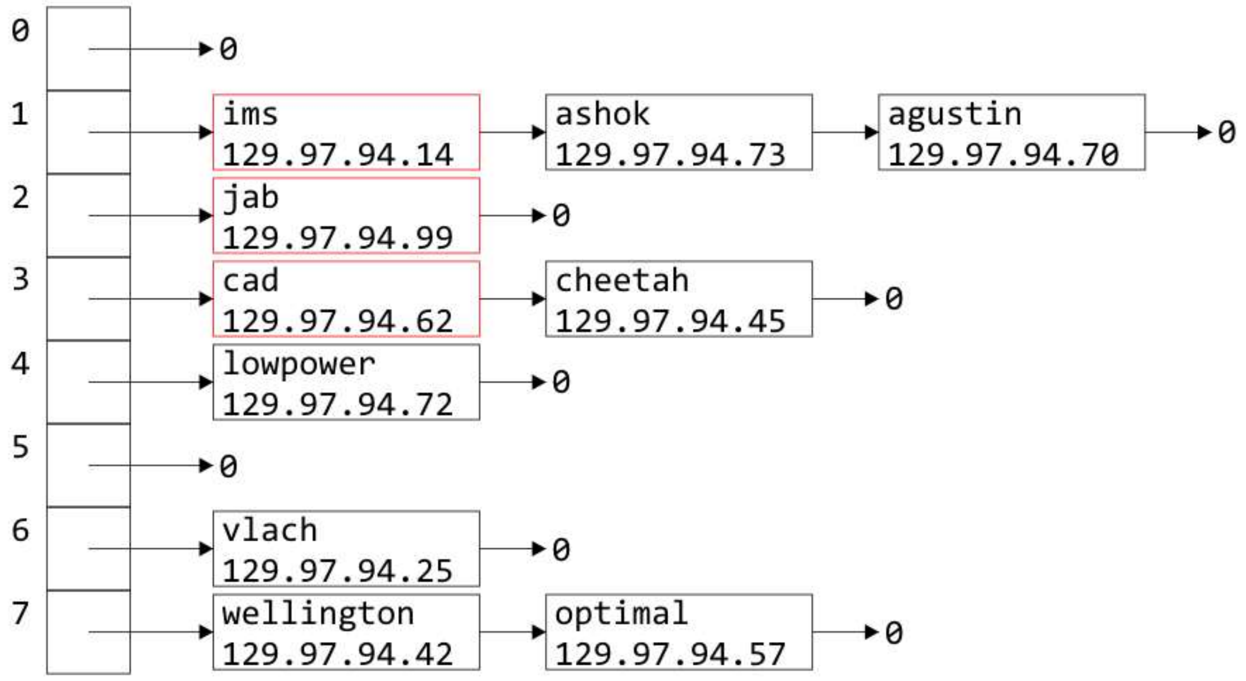
Example

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

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

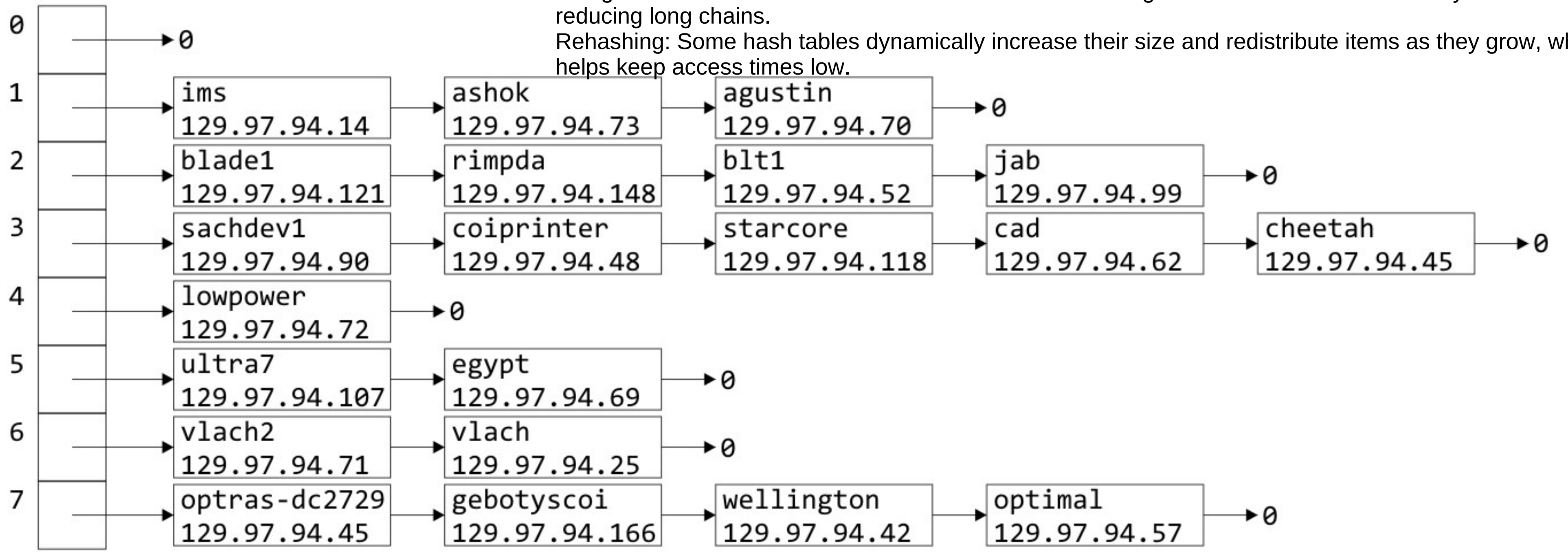


Example

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 $\mathbf{O}(k)$

Increasing the Number of Bins: Adding more bins could reduce the likelihood of collisions because there would be more places to store items.
Using a Better Hash Function: A different hash function might distribute items more evenly across bins, reducing long chains.
Rehashing: Some hash tables dynamically increase their size and redistribute items as they grow, which helps keep access times low.



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

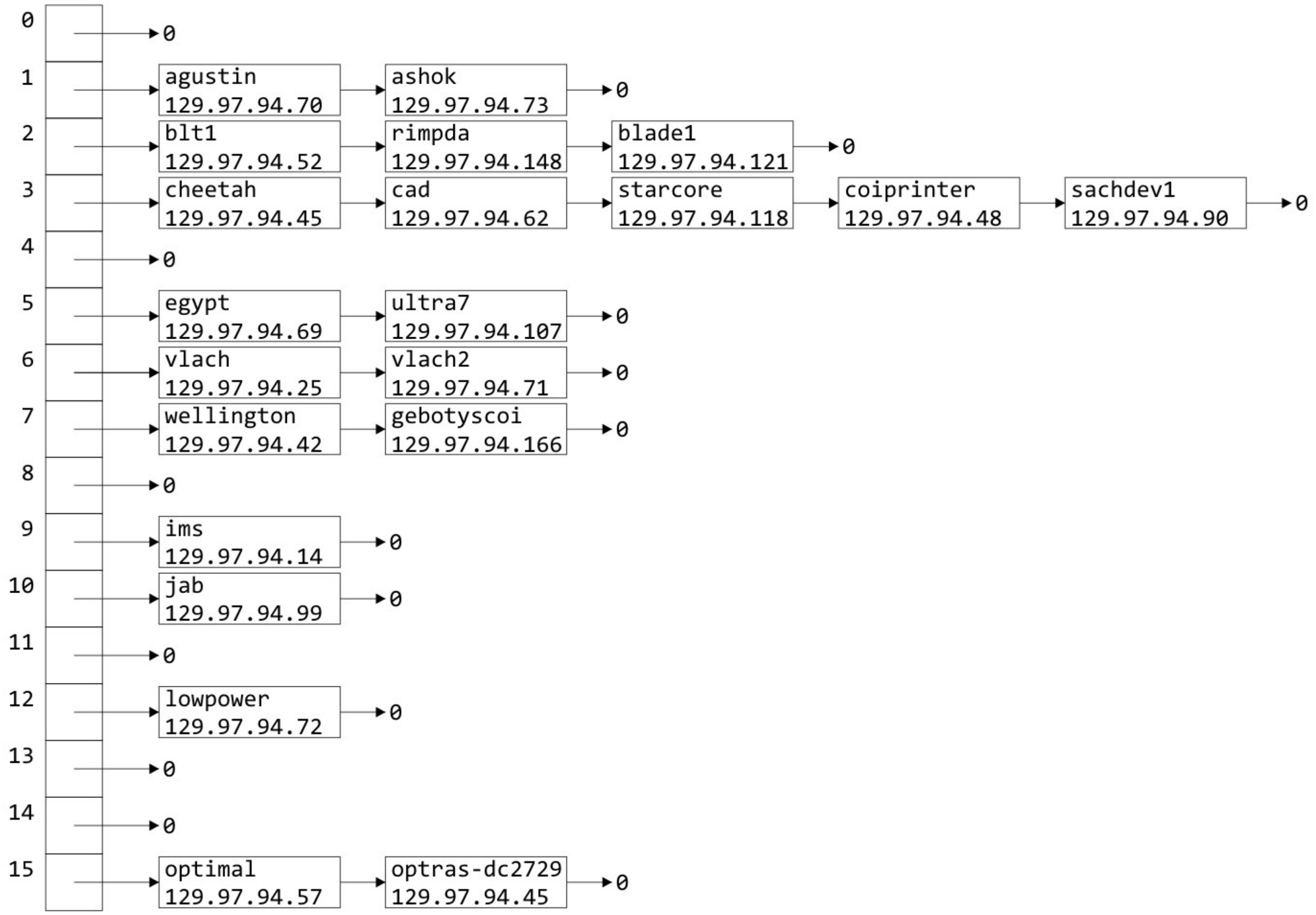
- Unfortunately, the hash function must now change
- In our example, the doubling the hash table size requires us to take, for example, the last four bits

Increasing the Number of Bins: Adding more bins could reduce the likelihood of collisions because there would be more places to store items.
Using a Better Hash Function: A different hash function might distribute items more evenly across bins, reducing long chains.
Rehashing: Some hash tables dynamically increase their size and redistribute items as they grow, which helps keep access times low.

Doubling Size

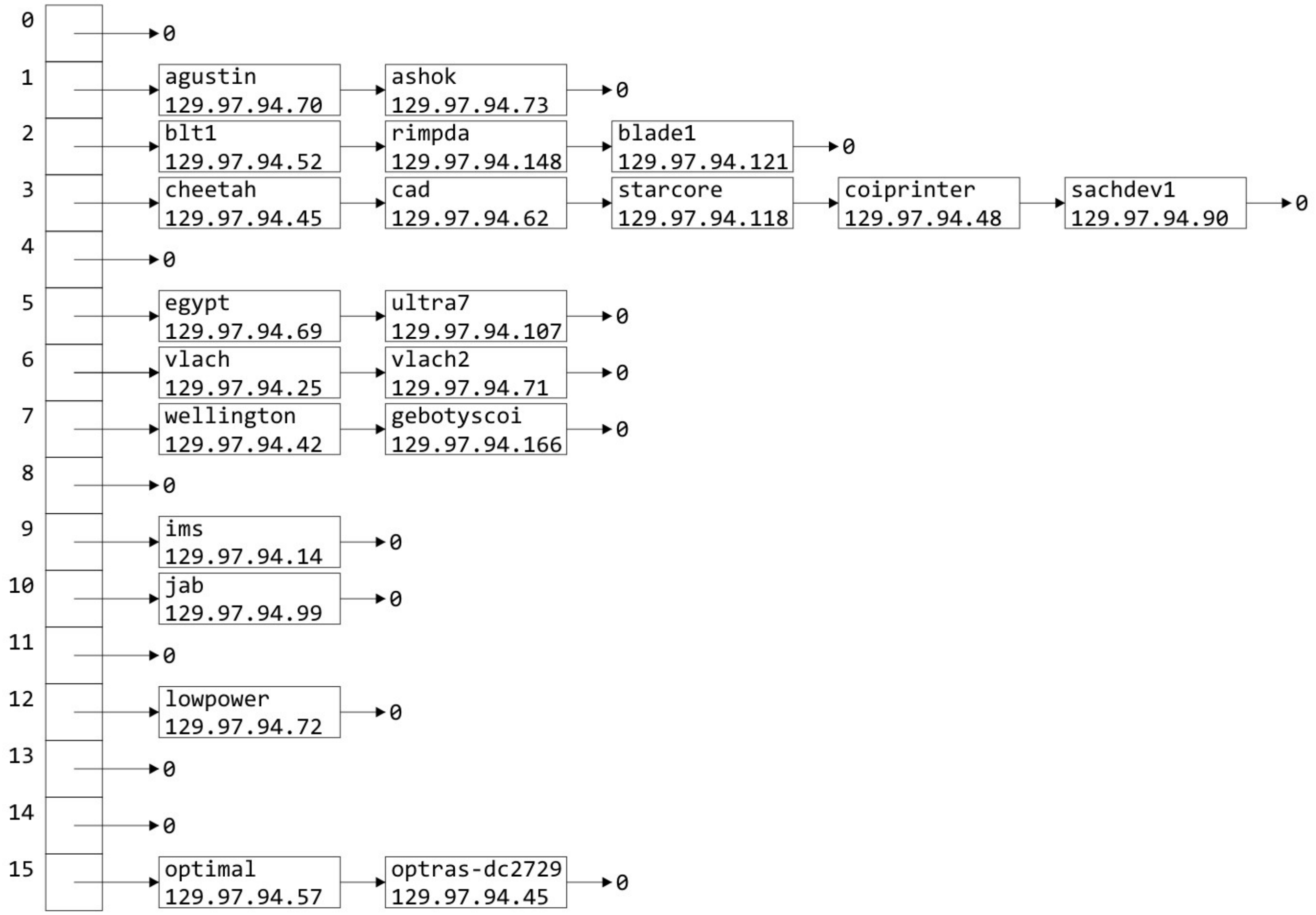
The load factor is now $\lambda = 1.3125$

– 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 Hash Function

We chose 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

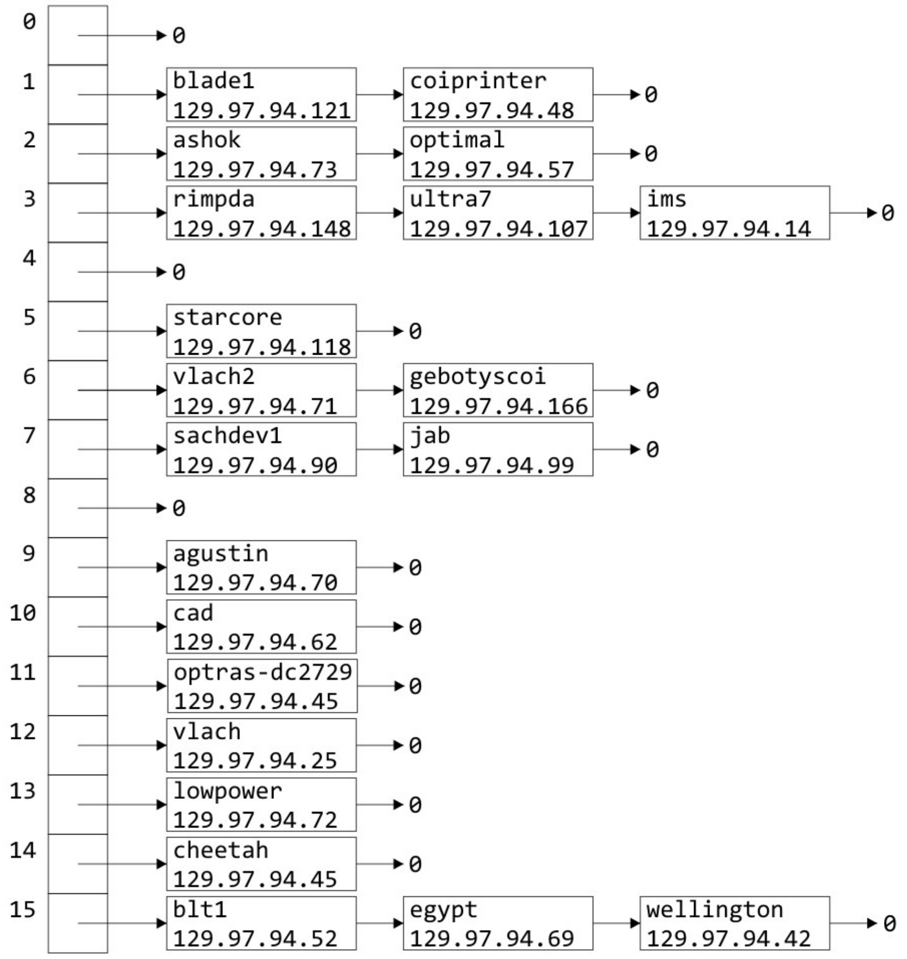
Choosing another Hash Function

Let’s look at the following hash function:

```
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 % 16;
}
```

Choosing another Hash Function

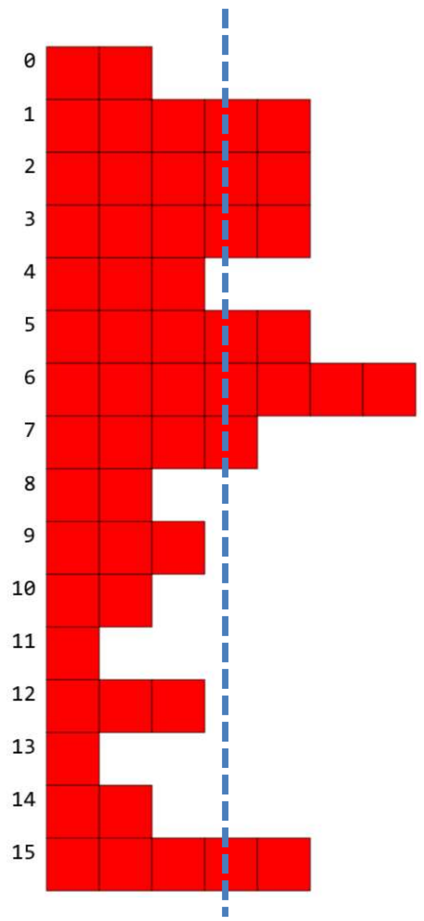
This hash function yields a much nicer distribution:



Choosing another 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…



A good hash function should distribute items uniformly across all bins to avoid clustering in specific bins. Here, we observe that some bins have more items while others have fewer or none, indicating that the hash function may not be distributing items evenly. In a well-balanced hash table, we would expect each bin to be close to the average load factor (3 or 4 items per bin) rather than having many bins with significantly more or fewer items.

Linked Lists in Chained Hash Tables

Issues 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 `Singly_list`)

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

Linked Lists in Chained Hash Tables

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

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

These slides cover the memory usage and a potential optimization for linked lists in chained hash tables.

Slide 1: Memory Issues in Chained Hash Tables

1. **Memory Usage in Linked Lists**:
   - Chained hash tables using linked lists have extra memory requirements due to the need to store pointers to next nodes in each list.
   - Dynamic memory allocation is used, which can lead to fragmentation and may be less memory-efficient compared to static arrays.

2. **Total Memory Calculation**:
   - **16 bytes**: Base memory for the hash table, which includes a pointer to the array, the initial and current number of bins, and the size.
   - **12M bytes**: This refers to the memory needed for `M` bins in the hash table, considering that each bin has a list. If each list in the `Singly_list` class tracks its size (or count), we need 12M bytes; without this, it’s 8M bytes.
   - **8n bytes**: If each object in the table is 4 bytes, storing `n` objects requires `8n` bytes (assuming an extra pointer or metadata overhead for each object).

Slide 2: AVL Trees as an Optimization for Linked Lists

1. **Replacing Linked Lists with AVL Trees**:
   - By replacing the linked lists with AVL trees, you can reduce the access time for each bin from \(O(k)\) (where \(k\) is the length of the linked list) to \(O(\log \lambda)\), where \(\lambda\) is the load factor (average number of items per bin).
   - This optimization assumes that you can order the objects, which allows for efficient searching and balancing with an AVL tree.

2. **Trade-Off**:
   - While AVL trees reduce access time, they increase memory usage. Each node in an AVL tree requires two pointers (left and right children), leading to an increase in memory by \(O(n)\) for storing these additional pointers.

Blackboard 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

References and Acknowledgements

- The content provided in the slides are borrowed from different sources including Goodrich's book on Data Structures and Algorithms in C++, Cormen's book on Introduction to Algorithms, Weiss's book , Data Structures and Algorithm Analysis in C++, 3rd Ed., Algorithms and Data Structures at University of Waterloo (https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials/) and https://opendsa-server.cs.vt.edu/ODSA/Books/Everything/html/BinaryTreeTraversal.html.
- The primary source of slides is https://ece.uwaterloo.ca/~dwharder/aads/Lecture_materials, courtesy of Douglas Wilhelm Harder.
- .