DS

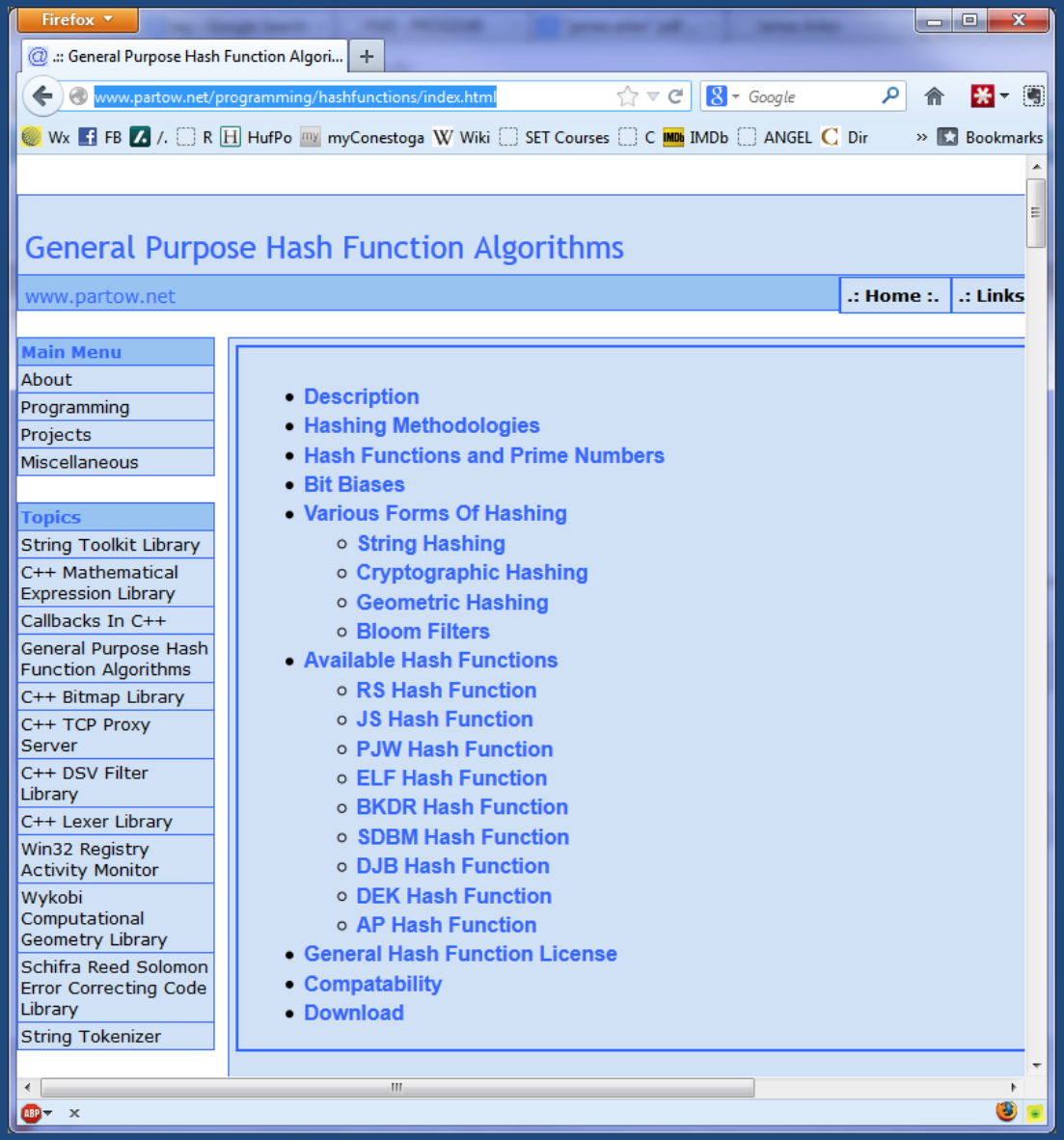
Hash Tables

Reading

http://en.wikipedia.org/wiki/Hashing_function

Reading

http://www.partow.net/programming/hashfunctions/index.html



Yes, this is over 100 slides long

But it'll be worth it!

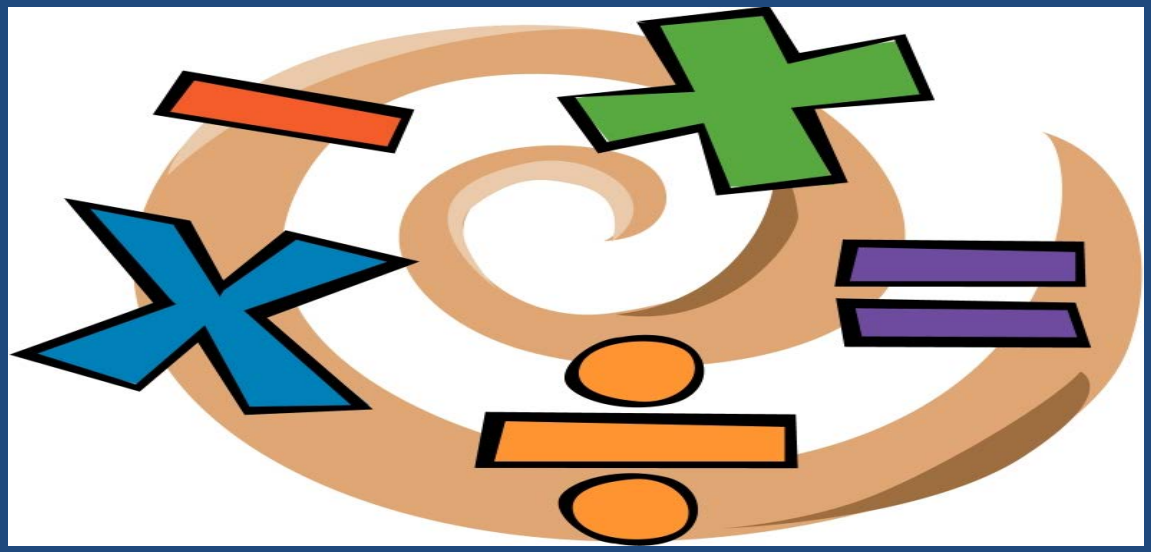
Hash Tables

The flexibility of
a linked list

+

the speed of access
of an array

Use a mathematical calculation



Chained Hash Table

An array of linked lists in which data is accessed through calculation on a "key"

Much more efficient, speed-wise, than using a normal array or a linked list

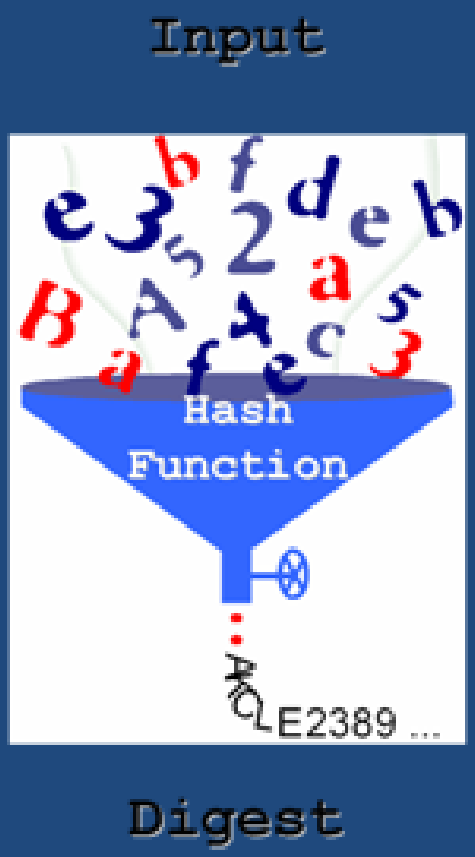
Aside

There are other types of hash tables

For now, we'll refer to "chained hash tables" simply as "hash tables"

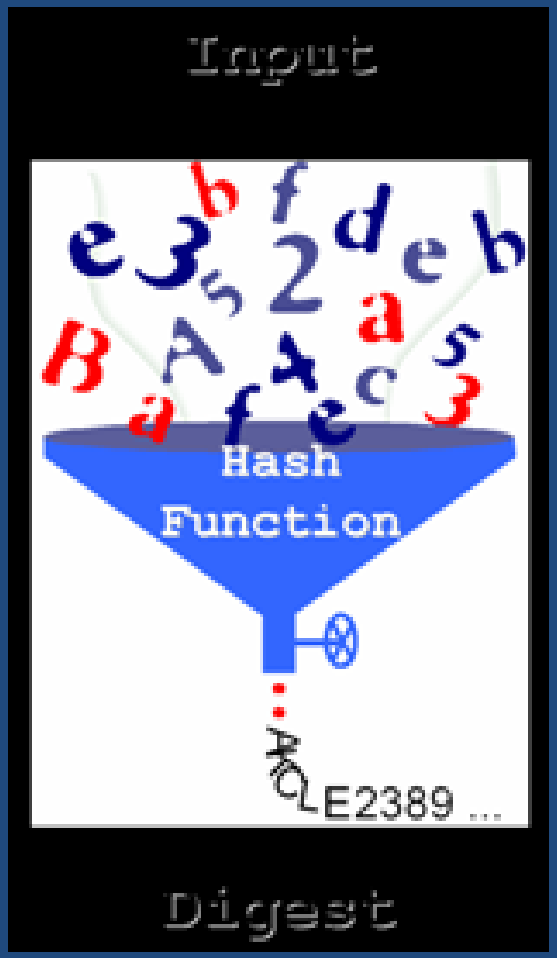
A “key”?

Data to be stored in the hash table is given to a special function called a “hash function”

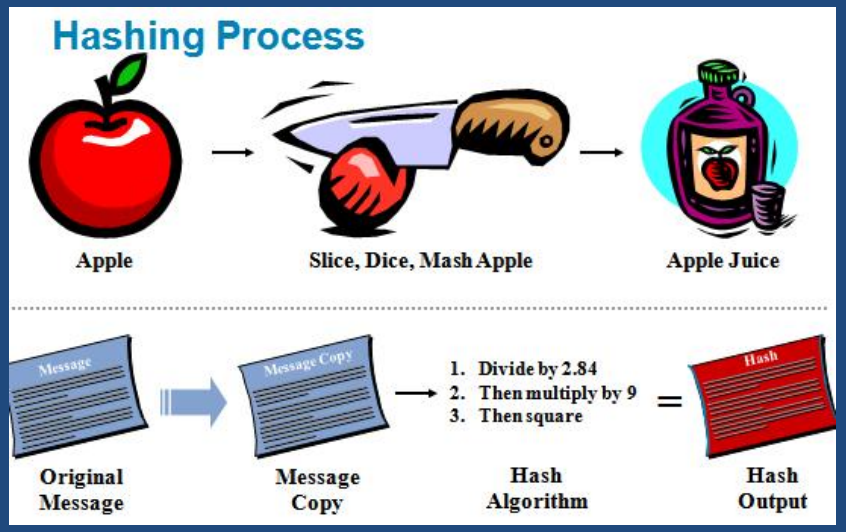


This data is called the "key"

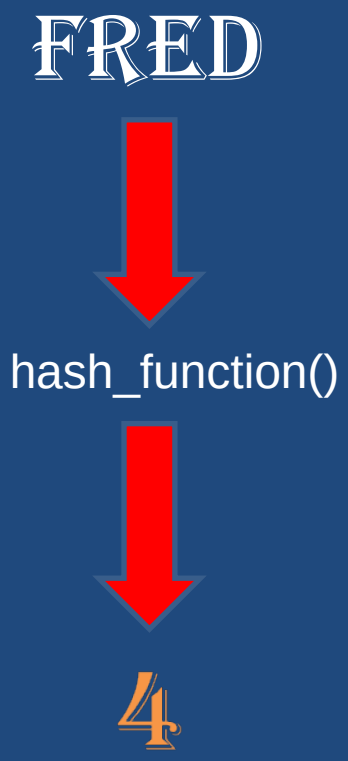
The hash function calculates a value based on the key provided to it



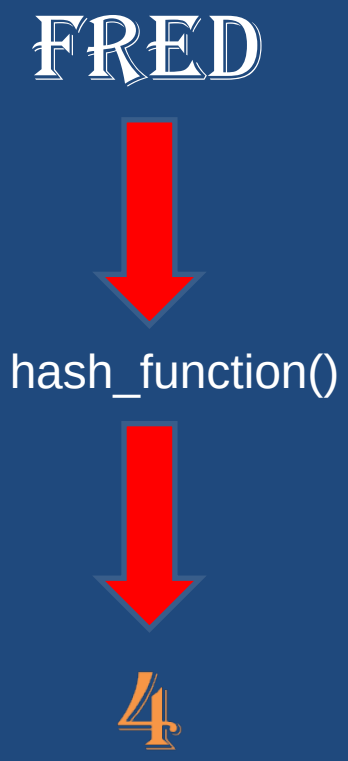
This value is called the "hash value"



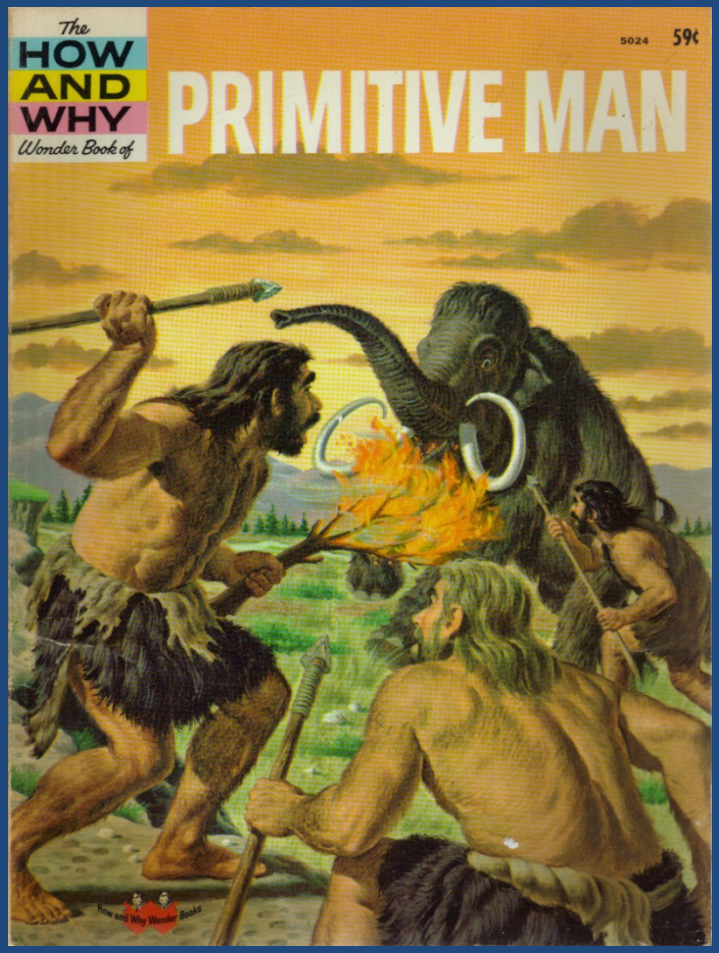
The calculation makes use of the data to generate a number



In this example, the function simply counts the letters in the data

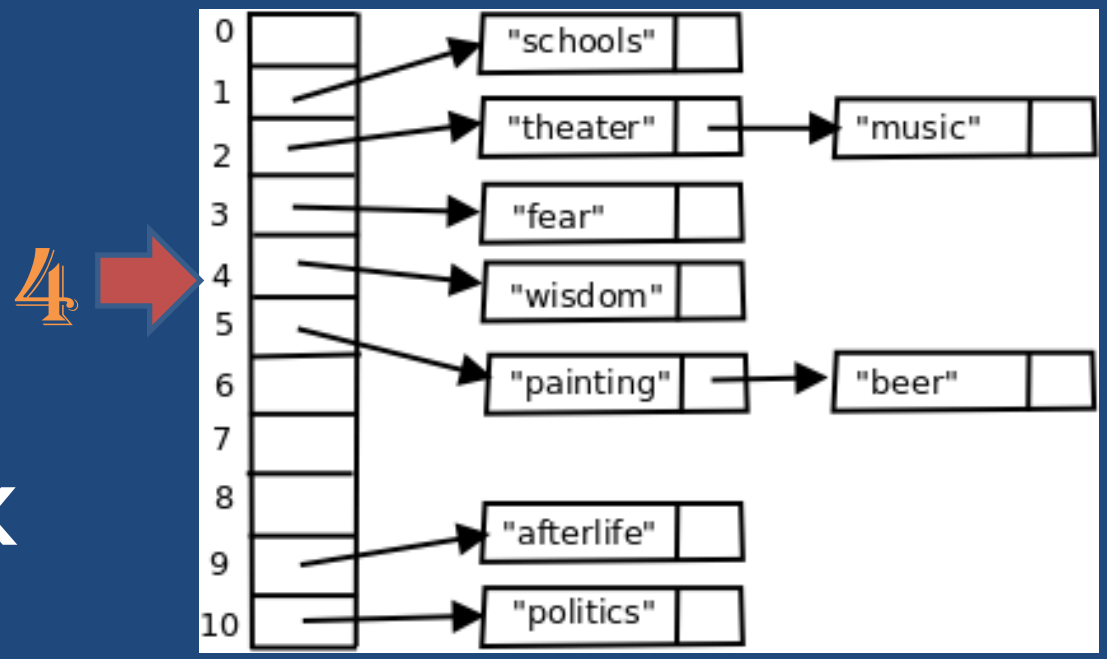


(that example was very simplistic and quite primitive and would never be used in real life)



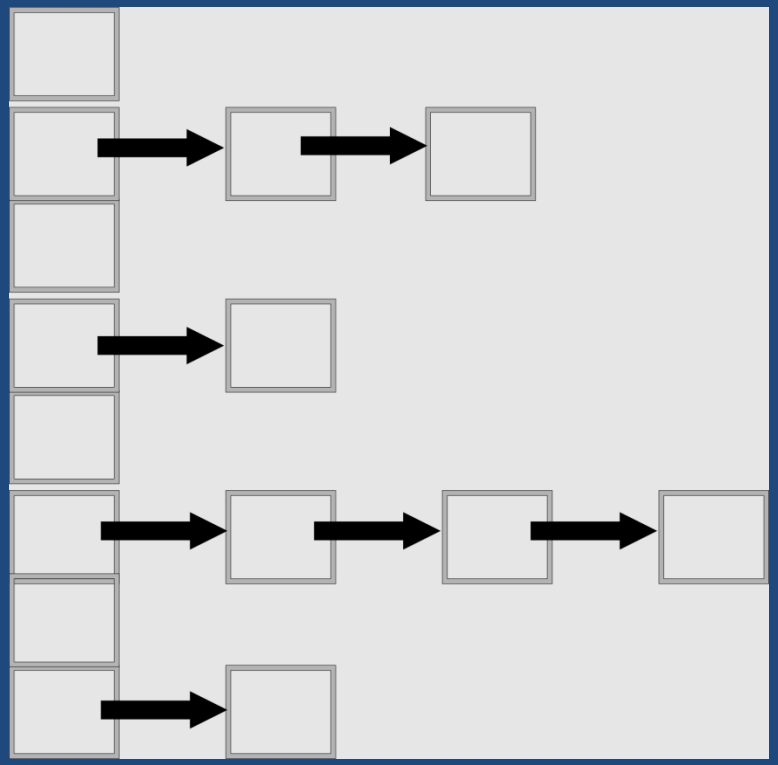
The hash value indicates where the data should be stored

The hash value is the array index



Use of the Array

An array of some fixed length is used to store a series of linked lists that contain data



Each array element is called a "bucket"

At first, each bucket has NULL, since there is nothing in the list

Illustration: An Empty Chained Hash Table

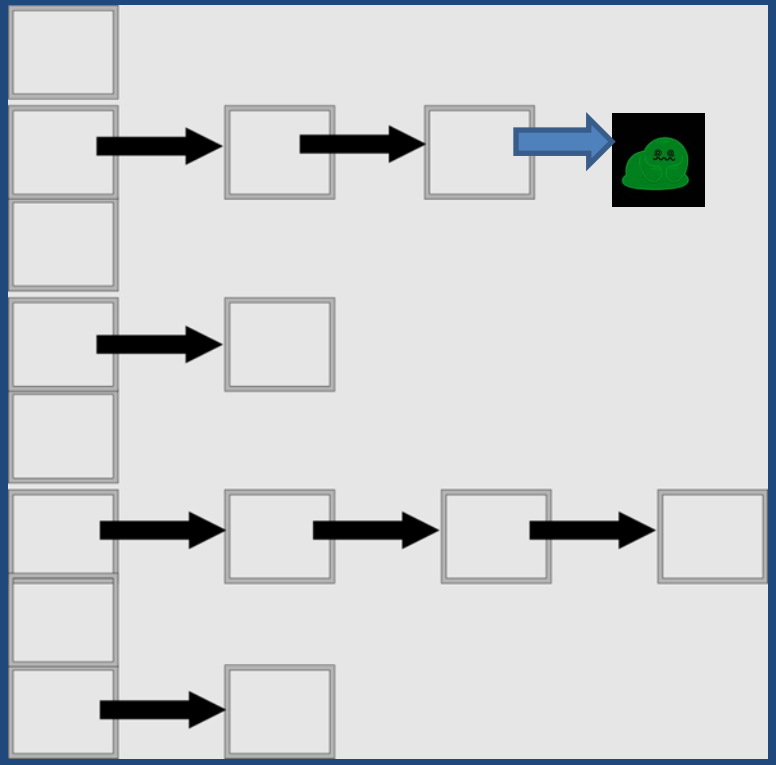
NULL
NULL
NULL
NULL
NULL
NULL
NULL
NULL

Adding an Item

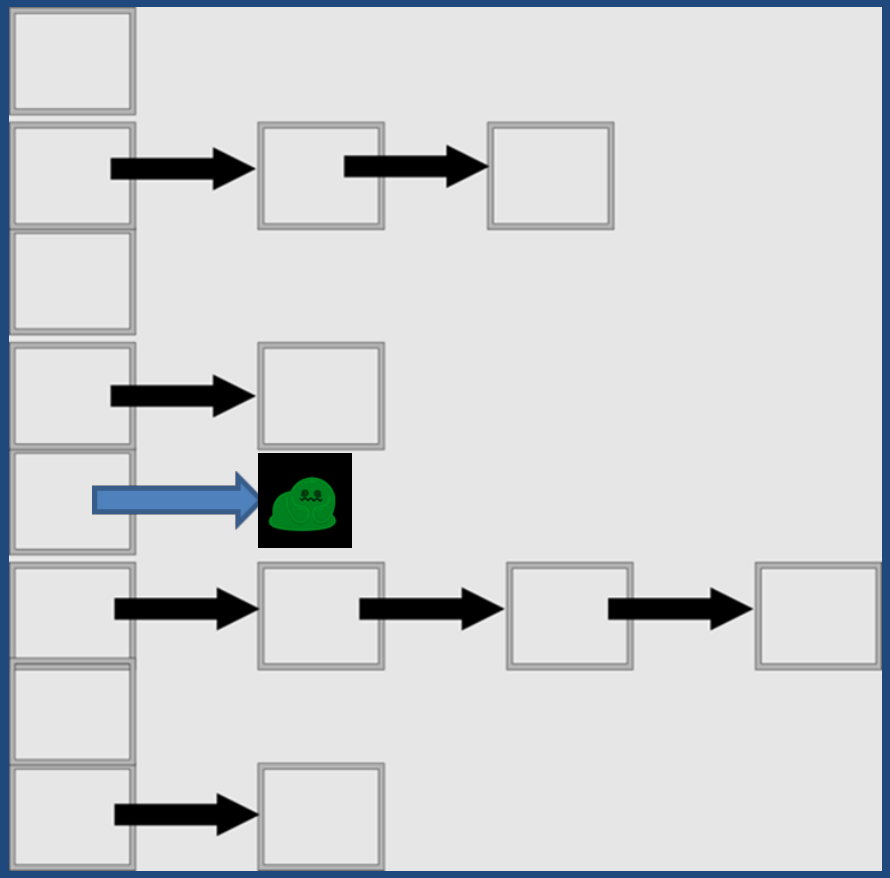
When an item is added, its key is passed to the hash function

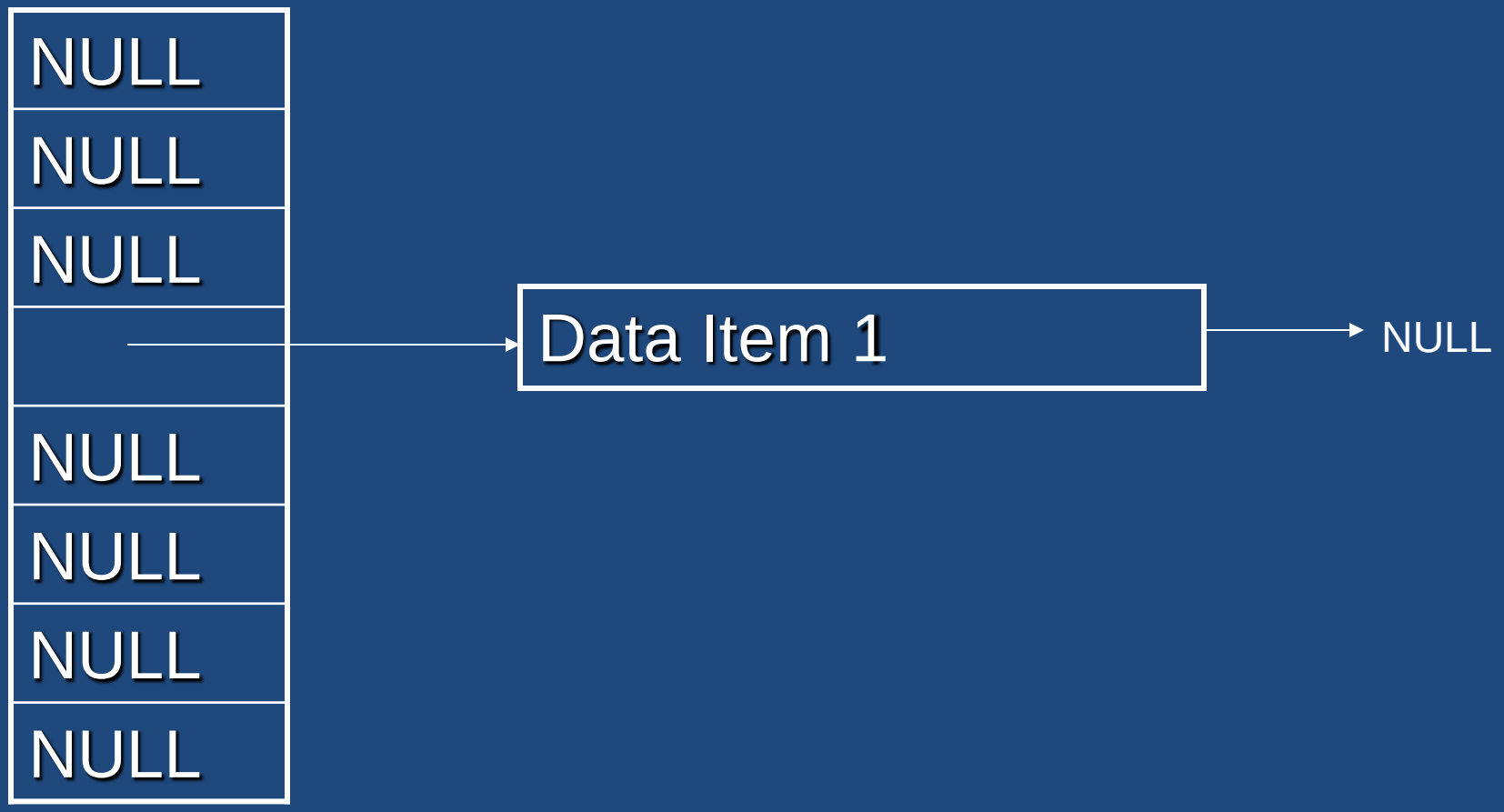
The hash function mathematically calculates which bucket the key should use for storage

If there is
already
a linked list
in that bucket,
we add
the item to it



If there is not a linked list in that bucket already, we create a new linked list in that bucket and add the item to the linked list

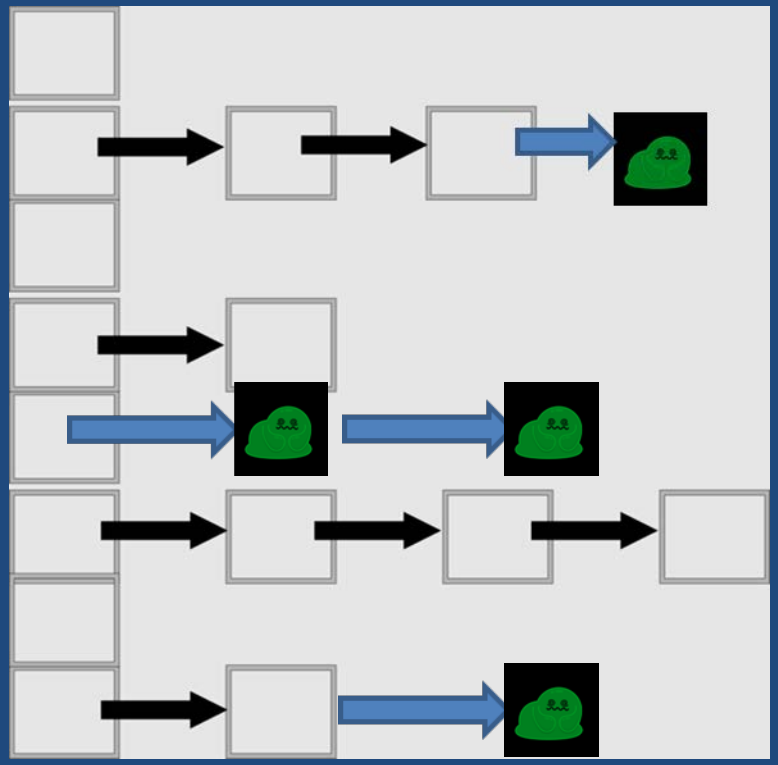


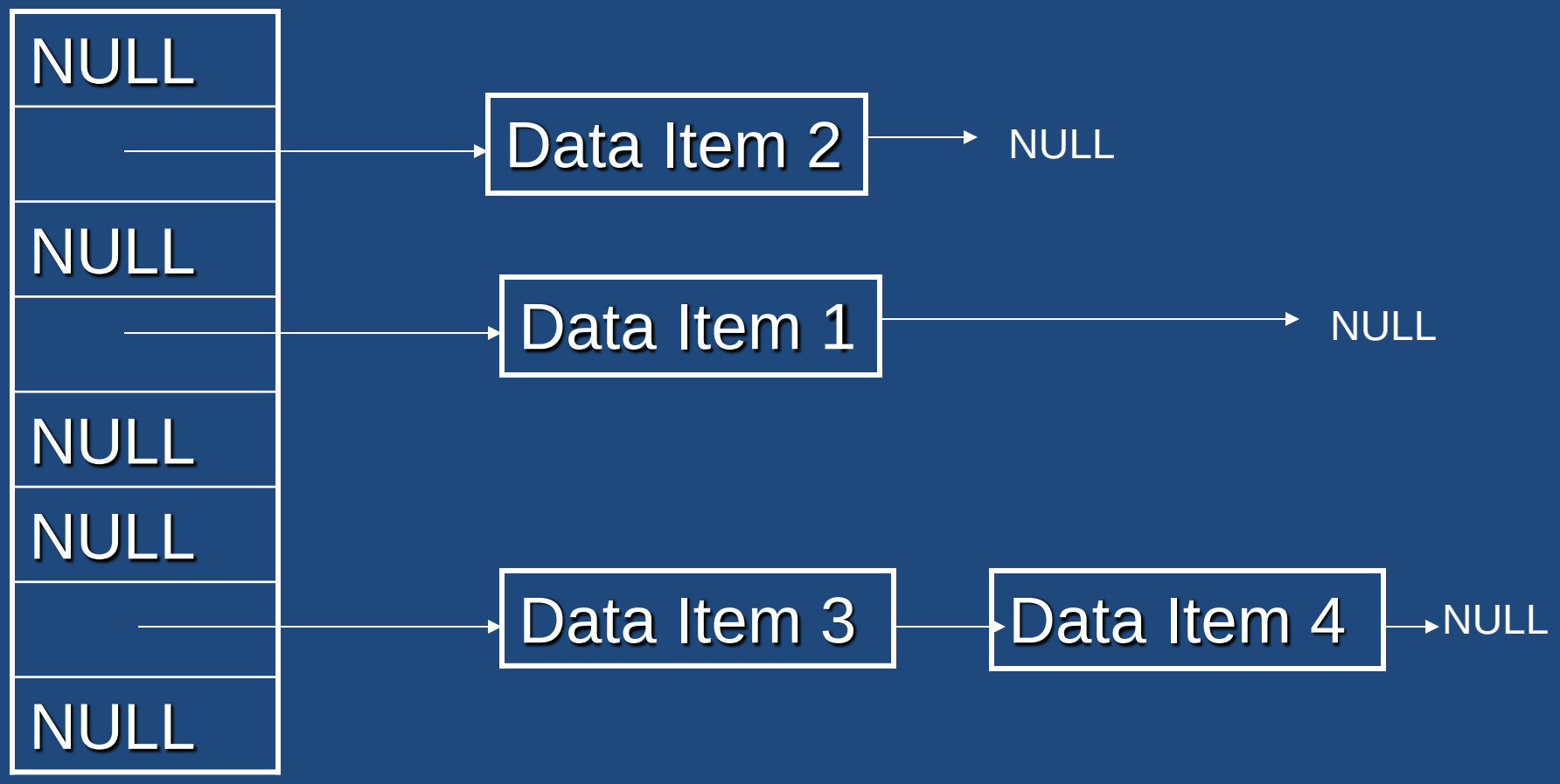

NULL
NULL
NULL
NULL
NULL
NULL
NULL
Data Item 1
NULL

Adding More

As we add more to the hash table, we calculate the hash values from the keys

Upon adding each item, we either create a new linked list at the bucket and add the item to it or we add the item to the already-existing linked list



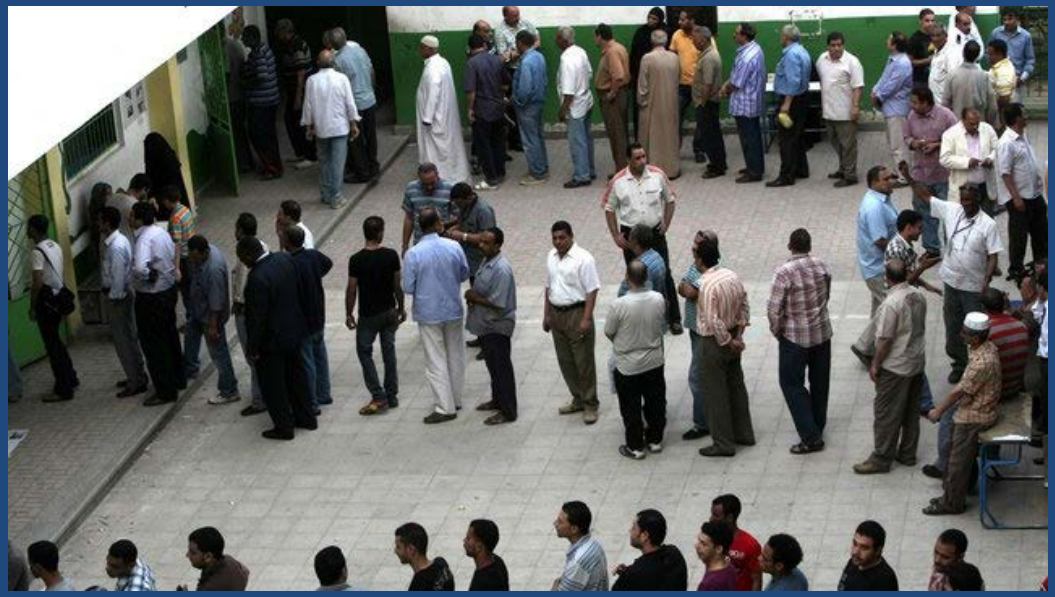

NULL
NULL
NULL
NULL
NULL
Data Item 2
NULL
Data Item 1
NULL
Data Item 3
Data Item 4
NULL

Keys and Buckets

The mathematical calculation in the hash function produces a hash value for each key

Typically, it is considered that the act of calculating the hash value is much faster than searching a linked list

Also, searching a short linked list is faster (on average) than searching a long linked list



Speed of Access

Finding your data
is fastest
if you had
many short
linked lists with
all of your
buckets filled

Ideally,
you would want
to have your data
distributed
as evenly as possible
(with respect to
how long your
linked lists are)

This is called “uniform hashing”

Advantages (Revisited)

"The flexibility of a linked list"

Each bucket's linked list can grow as large as you need

“... the speed of access of an array”

Calculation is done in the hash function so that you can find which bucket is used

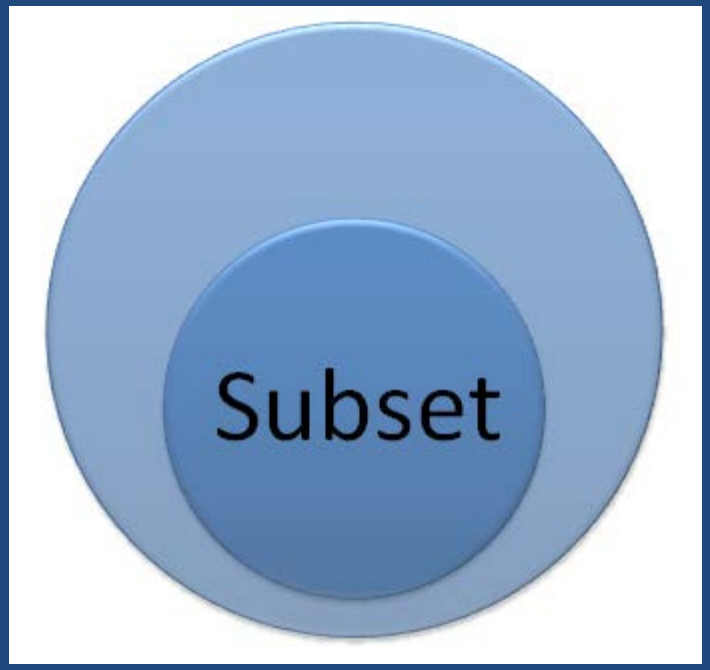
You then only have to search a much smaller set of data, consisting of the linked list found in that bucket

The Math of Hash Functions

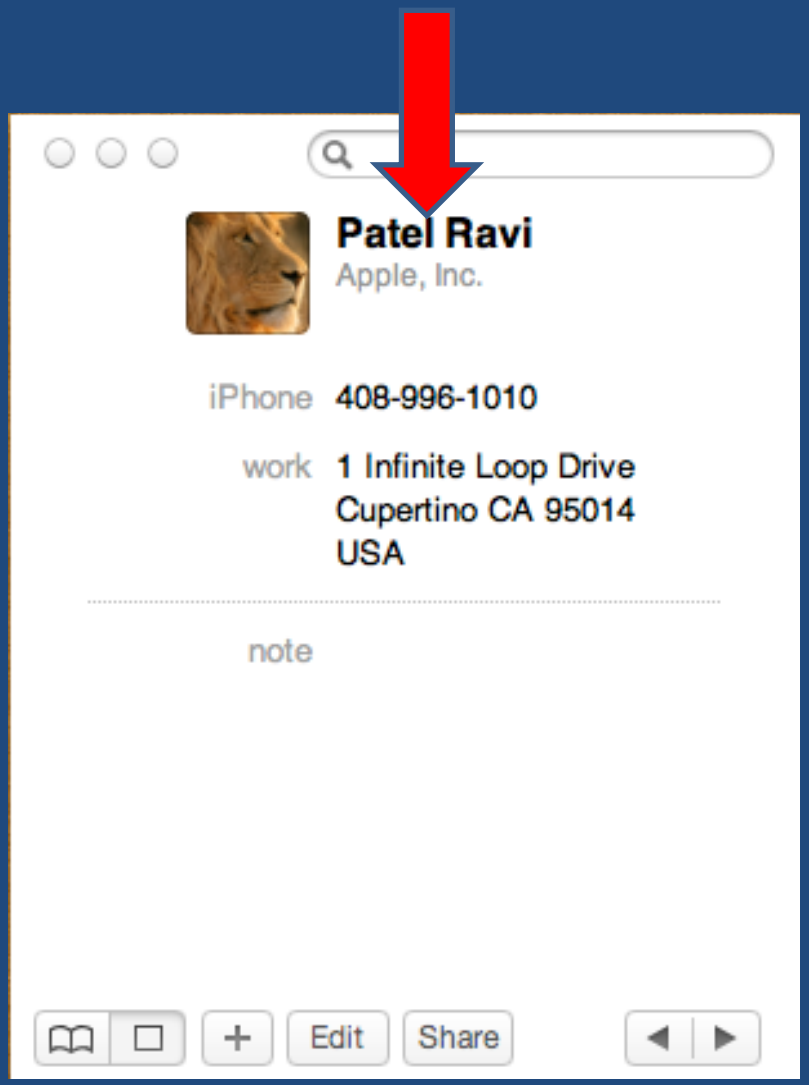
Again, ideally, you would want to have your data distributed as evenly as possible

The act of finding a hash function determines how efficient your hash table is

Your hash function acts on your data or on a subset of your data



e.g. if you are keeping address book information, you might use the person's full name as the key, rather than **all** of the information in the address book entry



A Trivial Really Bad Hash Function

Let's say that you are keeping track of address book information and you pass in the person's name as the key

"FRED"

An example of a really horrible hash function is the following:

```
int horribleHashFunction(char *name)
{
    return 4;
}
```

Why Really Bad?

The preceding example is really bad because it puts all of the data into one bucket

This has the effect of making your hash table, in essence, one long linked list

This hash function would never be used, since it's so useless, but it serves as an example of what you **could** do

But Don't!

Example of a Non-trivial Bad Hash Function

Let's say that you are keeping track of address book information and you pass in the person's **full firstname** as the key

Also assume that you have 8 buckets

An example of a bad hash function is the following:

```
int badHashFunction(char *firstname)
{
    return strlen(firstname) % 8;
}
```

Why Bad?

This example is bad because it heavily distributes the data into certain buckets

A Practical Illustration

According to the U.S. Social Security Administration data for the 1990's, the length of full firstnames were distributed as shown on the next page

Breakdown of Name Lengths

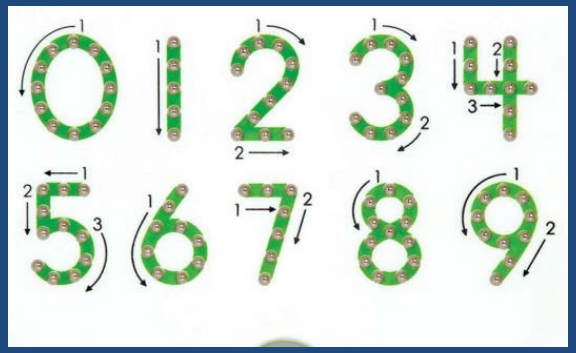
Length	Pct		Length	Pct
11 or 1	1.15%		6	27.44%
10	0.41%		5	22.24%
9	4.96%		4	11.16%
8	9.01%		3	1.10%
7	22.50%		2	0.03%

Let's use:

```
int badHashFunction(char *firstname)
{
    return strlen(firstname) % 10;
}
```

Why mod 10?

We choose to have 10 buckets
and
we need array index values from 0 to 9

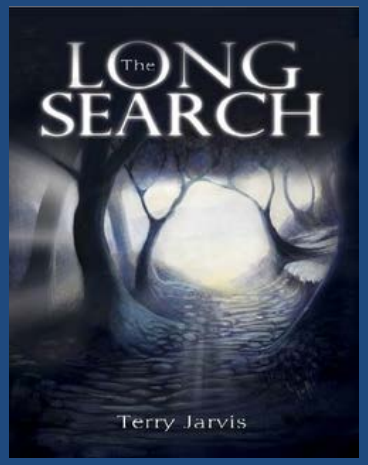


Analysis of Example

72% of the items go into buckets 5, 6, and 7

92% would get put into buckets 4, 5, 6, 7, and 8

This would result in a looooong search in most cases



Another Point about the Example

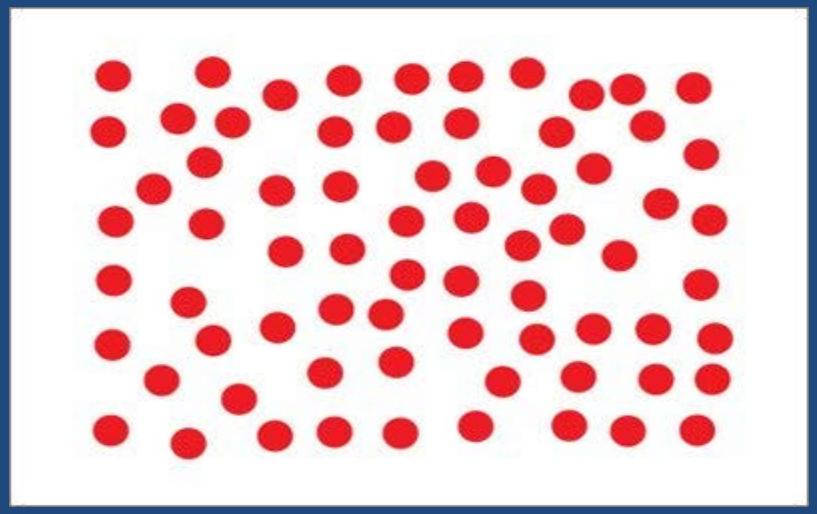
Note that data for length 11 and length 1 was grouped together

Both cases would return a hash value of 1, since the value was the key mod 10

Also, keys of length 10 would get put into bucket 0

Selecting a Hash Function

Distribute the elements as uniformly and without pattern throughout the hash table as is possible



This is “uniform hashing”

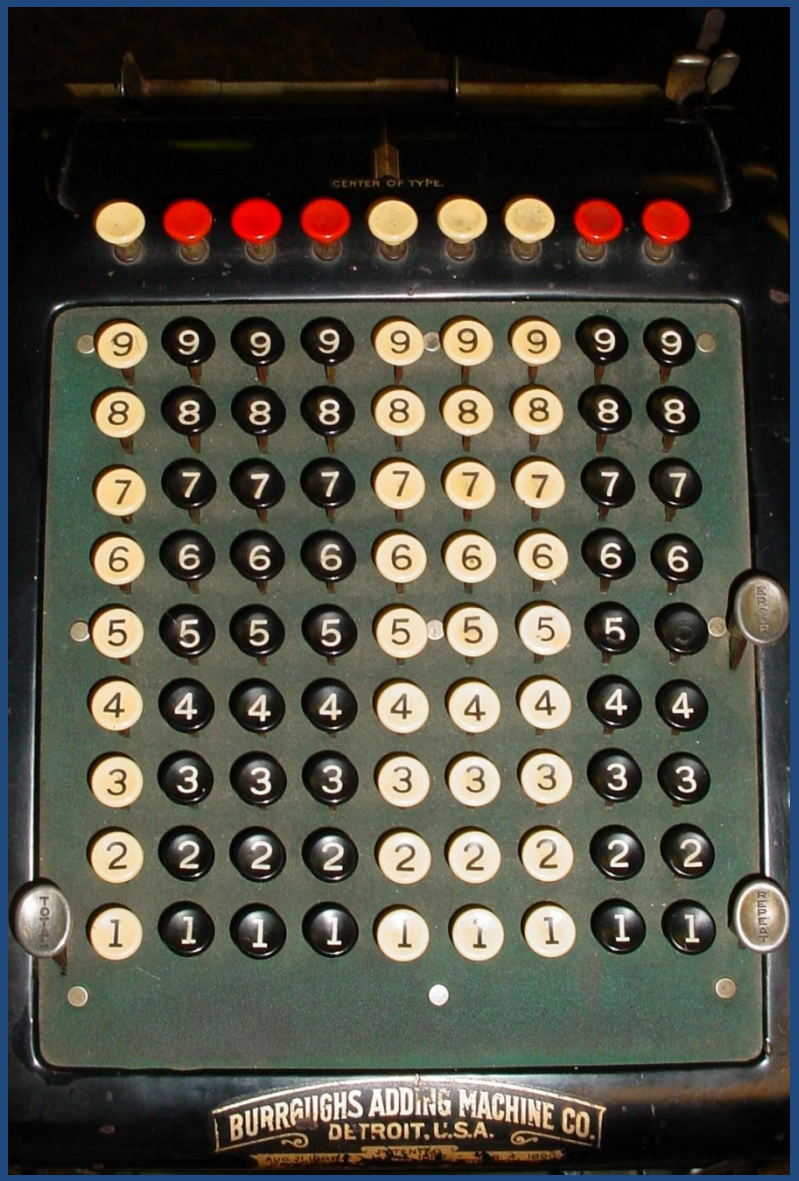
Transforming the Key

Typically, in a hash function, you transform the key into an integer

7

e.g. changing the string “fred” into the integer 4 by taking the length of the string

e.g. changing the string “fred” into the integer 417 by adding together the ASCII values of the four characters (i.e. ‘f’ is 102, ‘r’ is 114, etc.)



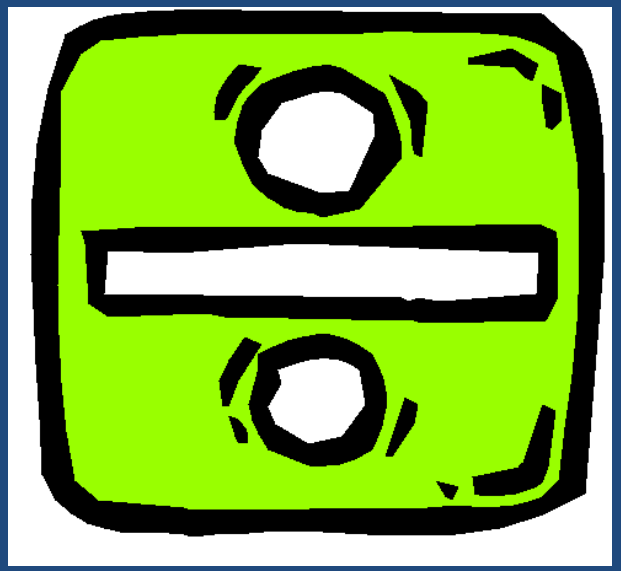
Generating the Hash Value

Then, you transform that integer (called the key value) into the hash value

The easiest way is to take the integer and use the mod operator, based on the number of buckets

e.g. hashValue = keyValue % bucketCount;

This is called the "division method"



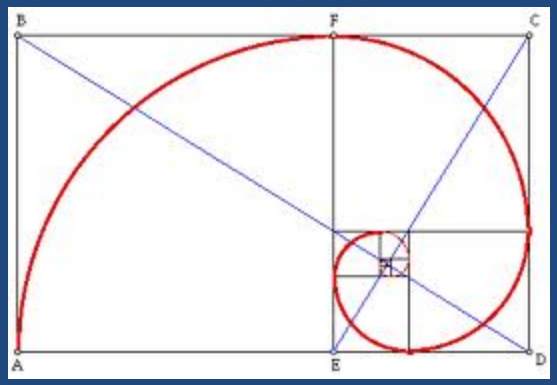
Another Way

Another way to generate the hash value is called the "multiplication method", which is:

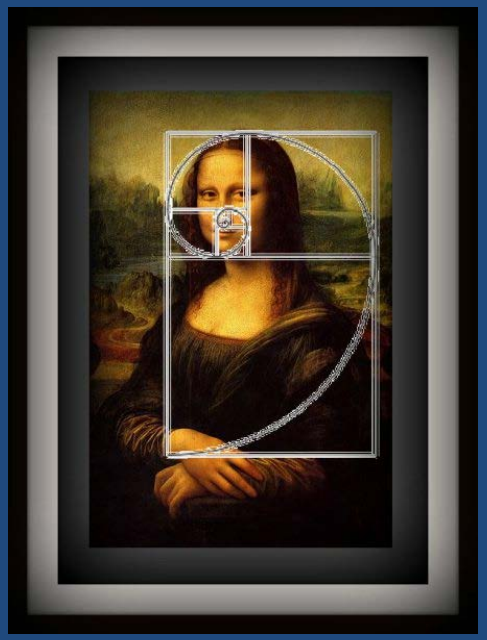
1. multiply the keyValue by a decimal value between 0 and 1
2. extract the fractional part
3. multiply the fractional part by the number of buckets
4. take the floor of the value (round down)

“a decimal value between 0 and 1”: The “Golden Ratio”

The “Golden Ratio” (approx. 1.618033) shows up over and over again in the real world, in geometry, nature, art, architecture, etc.



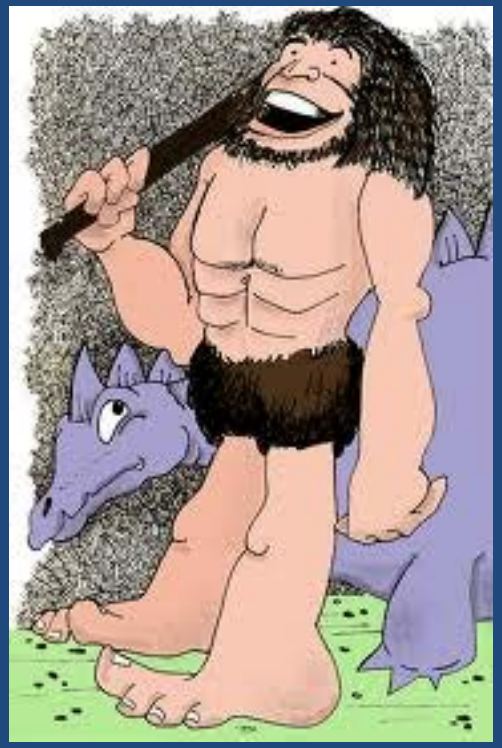
Donald Knuth’s The Art of Computer Programming, Volume 3 suggests that the inverse of the Golden Ratio (approx. 0.618) is a good choice for the multiplier used for hashing.



See
http://mathforum.org/dr.math/faq/faq.golden.ratio.html
or http://en.wikipedia.org/wiki/Golden_ratio
for more information about the Golden Ratio if you're interested in it

Other Hash Functions

There are many types of hash functions. We've only seen the most primitive ones so far.



If you're interested, take a look at http://www.partow.net/programming/hashfunctions/index.html for more details and examples for other hash functions.

(You don't have to worry about the math)

Other Applications of Hashing Functions

Hashing functions can be also used for:

- error correction
- audio
- passwords
- cryptography

Number of Buckets

If you're using the division method, your number of buckets should be a prime number

If you're using the multiplication method, your number of buckets should be a power of 2

The detailed reason for these is beyond the scope of this course

Load Factor

The "load factor" of a table is number of elements / number of buckets

Collisions

A "collision" happens when two keys hash to the same bucket

In a chained hash table, collisions aren't a problem, since you simply extend the linked list

Using a prime number as the number of buckets minimizes collisions

More About Hash Functions

From the Hash Function FAQ:

- http://burtleburtle.net/bob/hash/hashfaq.html

Examples

for (h=0, i=0; i<len; ++i) h += key[i]; is bad.

for (h=0, i=0; i<len; ++i) h ^= key[i]; is bad.

for (h=1, i=0; i<len; ++i) h *= key[i]; is very bad.

for (h=0, i=0; i<len; ++i) h = (h<<5)^(h>>27)^key[i]; is OK for one-word ASCII text, otherwise it's bad.

for (h=0, i=0; i<len; ++i) h = (h<<4)^(h>>28)^key[i]; is OK for one-word ASCII text, otherwise it's bad.

for (h=0, i=0; i<len; ++i) { h += key[i]; h += (h<<10); h ^= (h>>6); }
h += (h<<3); h ^= (h>>11); h += (h<<15);
is good.

for (h=0, i=0; i<len; ++i) h = tab[(h^key[i])&0xff]; may be good.

for (h=0, i=0; i<len; ++i) h = (h>>8)^tab[(key[i]+h)&0xff]; may be good.

Eh????

Many "good" hash functions are publically available.

You don’t necessarily have to understand **why** a hash function is “good”.

You should understand **how** to find and use a good hash function.

Understanding “bad”

In order to understand “good”, you must also understand “bad”.

The hash function FAQ describes certain hash functions as “bad” but ...

... they qualify that by stating:

"'Is bad" means probably OK for up to 256 buckets, but causes excess collisions beyond that.'

Cryptographic Hashes

All cryptographic hashes (MD5, SHA-256, Snefru, RIPE-MD) are good but complicated

and some are better than others

Firefox

Comparison of cryptographic hash func...

en.wikipedia.org/wiki/Comparison_of_cryptographic_hash_function

Wx FB R HufPo myConestoga Wiki ~set C IMDb Dir Home Dates Forms IP TripAdvisor Twitter D2L

Create account Log in

Article Talk Read Edit View history Search

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikimedia Shop

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools

Print/export

Comparison of cryptographic hash functions

From Wikipedia, the free encyclopedia

The following tables compare general and technical information for a number of cryptographic hash functions.[1]

Contents [hide]

1 General information
1.1 Notes
2 Compression function
2.1 Notes
3 Cryptanalysis
3.1 Notes
4 See also
5 Notes
6 References
7 External links

General information [edit]

Basic general information about the cryptographic hash functions: year, designer, references, etc.

Function	Year[gi 1]	Designer	Derived from	Reference
HAVAL	1992	Yuliang Zheng Josef Pieprzyk Jennifer Seberry		Website
MD2	1989	Ronald Rivest		RFC 1319
MD4	1990			RFC 1320
MD5	1992		MD4 RFC 1321 page 1	RFC 1321
MD6	2008			md6_report.pdf

Firefox

W Category:Broken hash functions - Wikip...

en.wikipedia.org/wiki/Category:Broker

Wx FB /. R HufPo myConestoga Wiki ~set C IMDb Dir Home Dates Forms IP

Create account Log in

Category Talk Read Edit View history Search

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikimedia Shop

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools

Print/export

Category:Broken hash functions

From Wikipedia, the free encyclopedia

Category for cryptographic hash functions that have been broken.

Pages in category "Broken hash functions"

The following 8 pages are in this category, out of 8 total. This list may not reflect recent changes (learn more).

G

- GOST (hash function)

L

- LM hash

M

- MD2 (cryptography)

M cont.

- MD4
- MD5

P

- Panama (cryptography)

S

- SHA-1
- Snefru

Categories: Broken cryptography algorithms | Cryptographic hash functions

This page was last modified on 24 August 2009 at 15:28.

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy.
Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

Privacy policy About Wikipedia Disclaimers Contact Wikipedia Developers Mobile view

a WIKIMEDIA project Powered By MediaWiki

Good Hash Functions: Paper

The author of the Hash Function FAQ has a paper on hash functions at:

- http://burtleburtle.net/bob/hash/evahash.html

He produces a hash function that he maintains is "good"

Unfortunately, the explanation of why it is good takes up six pages and involves a bit of math

There are many hash functions available on the Internet that are claimed to be good

- Best bet: try to find a hash function that claims to be good for the data that you're dealing with

A “claims-to-be-good” String Hash Function

Called djb2 (Created by Dan Bernstein)

```
unsigned long hash(unsigned char *str)
{
unsigned long hash = 5381;
int c = 0;
    while ((c = *str++) != '\0')
                hash = ((hash << 5) + hash) + c;
    return hash;
}
```

Bernstein now likes a modification of this:

- hash(i) = hash(i - 1) * 33 ^ str[i];
  - There is no adequate explanation why 33 works better than other numbers

You can use djb2
as long as you credit
where you got it

And there's

much,

much

more ...

ALL DONE

BLUE JAYS