• What’s a hash table?
• When to use a hash table
• When not to
• What is this hash function, anyway?

Updated: 2018-06-13
Hash? Like, the drugs or the potatoes?

Like a function that can map values of arbitrary length to values of finite length.

They’re useful for passwords. Passwords are text with arbitrary length. A hash function maps passwords to, lets say, a 256 bit integer value.

If somebody gets the hash, they have to brute force your password.
Hash Functions, Diagram Form

Password

"hunter2"

h(x)

2ab96390c7dbe3439de74d0c9b0b1767

Hash Function (MD5)

Hash (hexadecimal)
Wait, what class is this?

Sorry, still CS234. We won’t be hashing passwords. We still want hash functions though!

A Hash Table uses a Hash Function to store keys into an array **efficiently**

*average case*
Hash Map and Hash Sets

My diagrams show a Hash Set (it contains unique values). But you can easily add a second field to create a Hash Map (a dictionary) instead.

The concept is this: You have an array of “buckets”. The hash function tells you what bucket to use.

O(1) time complexity (hopefully)
Adding a value to a Hash Table

```plaintext
Test Value Please Ignore
```

```
333364597193746398350265891727706051329
```

```
% n → 1
```

```
buckets
len 8
```

```
h(x)
```

```
V
```

```
6
```
O(1)?

The hash function is O(1), and we can access any “bucket” in the array in O(1) time once we know the index.

So yes, O(1)
Simpler Keys, Simpler Hash

Keys will be natural numbers

Hash function is the identity function $h(x) = x$

CON: Not a good hash function (Python’s $\text{hash}$ does this though)

PRO: We can do this by hand
Another Example

Insert the usual values: 4, 8, 42, 16, 23, 15

1. Compute Hashes
   - $4 \mod 11 = 4$
   - $8 \mod 11 = 8$
   - $42 \mod 11 = 9$
   - $16 \mod 11 = 5$
   - $23 \mod 11 = 1$
   - $15 \mod 11 = 4$

2. Put Numbers in Buckets
What if something else wants to use that bucket?

This is called a “collision” and it’s bad. You can’t have two things in a bucket.

The solution (ok, A solution) is something called “Open Addressing”. That means “A value can go at any index, but the hash function gives you a starting point”
Open Addressing

The hash function tells us the first location we can use to store an item. If not, we check the “alternate locations”

This is called “probing” because we’re probing for a free bucket.

To do this, we’ll need a probe function $p(i)$ that tells us what alternate location $i$ will be (the start location is “alternative #0”)
Linear Probe

The simplest probe is the linear probe

\[ p(i) = h(x) + i \]

It’s like a linear function, \( y = ax + b \)

\( a = 1, \ b = \text{hash} \)
Linear Probe Example

\[ h(0) \equiv 0 \pmod{8} \]

\[ h(8) \equiv 0 \pmod{8} \]

\[ h(16) \equiv 0 \pmod{8} \]
Runtime for Insert

If something is already in bucket $h(x) \% n$, linear probe may need to check $O(n)$ buckets before it finds a free bucket!

So, $O(n)$. This could happen even if there is only one collision.
Run-Time for Lookup

Worst-Case runtime: $O(n)$, as we have degraded to linear search.

This happens no matter what, even if there are no collisions during the inserts (just with the search)

E.g. if we insert $(0,1,2,3,4,5,6,7)$ then the whole array of length 8 is full, and searching for 8 will fail only after searching the whole array
Time Complexity, IRL

CS116 promised $O(1)$ didn’t it? Well, actually it promised “basically $O(1)$”.

Assume keys are chosen uniformly at random.

Assume “sufficiently large $n$” is 297,467

(That’s a prime number congruent with 3 modulo 4)
Linear Probe Plots

average cluster size
Linear Probe Plots (2)
Boy, that escalated quickly!

It started out pretty much flat, but things really got out of hand fast!

When you’re close to capacity, there are large clusters. Unavoidable.

Solution: Don’t get close to capacity
Load Factor

For a hash table, the *load factor* (usually represented by the Greek letter $\alpha$) is the number of elements divided by the capacity.

That is, if you have 8 elements in a hash table with 16 buckets, the load factor is $\alpha = \frac{8}{16} = 0.5 = 50\%$

If we resize when $\alpha$ exceeds 66%, we can avoid the spike in cluster sizes.
Resizing a Hash Table

Each element was placed according to $h(x) \mod n$. Changing $n$ will change this!

You have to **rehash**. Meaning, make a new table, and insert all $n$ values, one at a time.

If we assume $O(1)$ average case for inserts, this makes hash table doubling $O(1)$[amortized+average case].
Deleting Values

Let take \( n = 8 \), and insert values 0,8,16:

\[
\begin{array}{cccccccc}
0 & 8 & 16 & \circ & \circ & \circ & \circ & \circ \\
\end{array}
\]

OK, now delete 0

\[
\begin{array}{cccccccc}
\circ & 8 & 16 & \circ & \circ & \circ & \circ & \circ \\
\end{array}
\]

Now search for 8. \( h(x) \% 8 == 0 \)

\[
\begin{array}{cccccccc}
\circ & 8 & 16 & \circ & \circ & \circ & \circ & \circ \\
\end{array}
\]
Deleting Values

Solutions?

We could find where any collisions with 0 got put, and move them around. That sounds like O(n) to me

(In fact you only need to scan within the current cluster for linear probe, but for other probe strategies it will require scanning the entire table)
Deletion Markers

To Delete 0 we can mark 0 as deleted but keep it there.

Insert will count “deleted” as “empty” but search will count it as “full”

That way, if we insert, say, 32, it will be placed at index 0, but searching for 8 will still succeed.
Deletion Markers

I used \( \nabla \) (a “nabla”) for the “DEL” symbol because: Mathematics calls it the “del” operator (it’s not actually short for “Delete” but it’s too convenient to pass up)
Deletion Markers and Load Factors

Should a DEL marker count against the load factor?

Do you want to ensure fast searches? DEL should count.

Drawback: You might resize when there’s only one value in your array, but the array length is already 512!
Good Fences

An issue with clusters is that bucket 1’s probe pattern is 1,2,3,4,…

While bucket 2’s pattern is 2,3,4,5,…

That’s a lot of overlap. Would be nice if their patterns didn’t overlap so much!
Quadratic Probe

New Probe Pattern:
\[ p(i) = i^2 + h(x) \]

i.e. the probe offsets will be 0, 1, 4, 9, 16, …
Quadratic Probe (cont.)

With linear probe, \( n \) probes is going to find a free bucket if there is one. Is Quadratic?

Maybe.

With \( p(i) = i^2 + h(x) \), this is not guaranteed to work.

If \( n \) is a prime number, the first \( n/2 \) probes will be unique (modulo \( n \)). Pretty bad. (We would need to resize when the load factor is only 50%)
Why I chose a prime length

It turns out that alternating the sign lets us merge the two “half the unique values”:

\[ p(i) = h(x) + (-1)^i i^2 \]

n probes will hit all n unique values (modulo n) but this is only a guarantee when n is a prime number, and \( n \% 4 == 3 \)
Quadratic alternating probe

Let $n$ be 11 (that’s a prime and $11 \mod 4 == 3$)

Consider $h(x) = 1$, $h(y) = 2$. Neighbours.

X’s probe pattern for the first 6 attempts is:

$1, 0, 5, 3, 6, 9…$

Y’s probe pattern is:

$2, 1, 6, 4, 7, 10…$

If capacity is < 55%, they only probe one common index.
Quadratic Alternating Sign

average probe count

![Graph showing the average probe count for quadratic alternating sign with x-axis values from 0 to 300000 and y-axis values from 0 to 5.](graph.png)
Hey, I was promised better!

It is better. Never believe a graph without comparing the scales! Here, let’s see…
Linear vs Quadratic, Largest Cluster
Quadratic Probe

Advantage

• The spike happens later. We can wait until $\alpha > 80\%$ to rehash!
  • Oh, but it doesn’t work when $\alpha > 50\%$ unless you have a prime length.
  • Not a problem, we can precompute the lengths

Disadvantage

• Computers like $\%n$ when $n$ is a power of 2. They do not like it when it is a prime.
Other Constants

Quadratic: \( f(x) = ax^2 + bx + c \)

If \( a = \frac{1}{2}, \ b = \frac{1}{2} \) then you have \( p(i) = \frac{i(i+1)}{2} + h(x) \)

This is a special case, the **triangular numbers**.

In general this does not hit every bucket without repeats

If does, however, work if \( n \) is a power of two.
Conclusion

Use Powers of Two, use Triangular Numbers

I’d show you a graph but there is no visible difference between quadratic alternating sign and the triangular numbers
Collisions (take 2)

Even with this new probing technique, collisions hurt.

\[ X = 0, \ Y = 11 \]
\[ h(x) = h(y) \]

Their fancy quadratic probe pattern will be identical.
Double Hashing

Another approach we could take is to use a second independent hashing function, h’

\[ p(i) = h(x) + h'(x) \times i \]

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>h(x)</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>h'(x)</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Probe</td>
<td>0,8,5,2,10,7,4,1,9,6,3</td>
<td>0,2,4,6,8,10,1,3,5,7,9</td>
</tr>
</tbody>
</table>
Does that always work?

No. What if \( h'(x) \equiv 0 \pmod{n} \)?
\[ p(0) = h(x), \ p(1) = h(x), \ p(2) = h(x) \ldots \]

Solution:

\[ p(i) = h(x) + i \cdot (1 + (h'(x) \% (n - 1))) \]

Now the coefficient will be between 1 and \( n-1 \).
Does it always probe unique values?

Maybe?

Double hashing uses a linear function, where the slope is obtained from the secondary hash function.

The first $n$ values of $f(x) = ax + b \ (\text{mod} \ n)$ will be unique if $a$ and $n$ are mutually prime

Once again, we want the length to be prime
Another Way to be (mutually) prime

What are the prime factors of a power of 2?
JUST 2.

How many times does 2 divide into an odd number? None! That’s what odd means.

So, instead of messing with primes:

\[ p(i) = h(x) + (2 \times h'(x) + 1) \times i \]
Quadratic vs Double (Primes)
Quadratic vs Double (Power of 2)
Looks like a tie?

They seem pretty close, right? Well, we’re using uniform values. What if I change my distribution:

key = (random number) * n

That’ll always be congruent to 0. Collision city. But only for h(x), not h’(x)
Worse Case (double hash)
Worst Case (linear or quadratic)

In the worst case, Open Addressing is $O(n)$ (the probe hits every full bucket).

300,000 is a fair bit larger than 6, so the comparison graph would be silly.
Wait, did the double hash slide say “Worse”? Yeah. Worst case scenario is when all keys are equal for both hash functions (double hash collision).

While I was using identity for $h(x)$, I used MD5 for $h'(x)$.

Not many values will be congruent for BOTH hash functions.
Worst Case Scenario

Python (and many other languages) use Hash Tables to implement the Map/Dictionary ADT.

If you know what hash functions are used, you could compute worst case strings and force them all into the same bucket. Bad news. User choice invalidates our statistics!
Random vs. Adversary

If the hash function(s) is/are chosen randomly, then the adversary cannot know what the worst case input is, so we’re safe to assume it’s a statistical improbability.
What’s the other approach?

The alternative to “Open Addressing” is “Separate Chaining”.

(It’s also called “Open Hashing” which is just confusing).

The idea here is: Who said we can’t have two values in one bucket?
Separate Chaining

Instead of an array of values, the table could be an array of linked lists.

If the Hash function puts two values in one bucket, we have two things in a linked list. No Problemo.

This is called ‘separate chaining’ because each bucket has a separate linked list (a chain of nodes).
Example Do-Over

Wait, aren’t those arrays? I thought you said linked lists?!

Just because it’s called chaining doesn’t mean you are beholden to the linked list. I can array if I want to.

I could use a tree, too.
Separate Chaining (cont).

Could we use another hash table as the contents of the buckets?

No, that’s absurd!

(Yes, we could, but we’d want to use a different hash function, and we couldn’t just have it be hash tables all the way down)
Creating a Hash Table

Function HashTable(n):
    cap = n
    len = 0
    bucket = Array(n)
    for i = 0 to n-1:
        bucket[i] = List()
Adding to a Hash Table

Function Add(H, value):
    key = hash(value) % H.cap
    if value not in H.bucket[key]:
        H.bucket[key].append(value)

Time Complexity?

Worst Case: $O(n)$ (because all $n$ values are in the same bucket)
Time Complexity, IRL

Average and Max linked list lengths vs. n (number of values in the table).
Average Case

If you look at the average case cost (this is the average length of each non-empty linked list in the hash table), it’s growing very slowly.

Conclusion: It’s basically $O(1)$. Just look at it! OK it’s not perfectly flat, but nobody’s perfect.

The longest length looks like it’s $O(\log n)$ too! Maybe?
Proof by Visual Inspection

We really should do Math instead of just eyeballing it. But we won’t.

As $n$ goes to infinity, the expected length approaches $1/(1 - 1/e)$ approx. 1.582

The odds of diverging substantially are basically 0%
Load Factor with Separate Chaining

There was no spike as we approached 100%

With multiple values per bucket, separate chaining can have a load factor greater than 100%

You will still want to resize, but the threshold can be 150% instead of 50%-75%
Summary

Separate Chaining:
Best Case: Lowest constant
Worst Case: $O(n)$ when hash function collides

Linear Probe (double hashed):
Best Case: Close to Separate Chaining
Worst Case: $O(n)$ when both hash functions collide (or nearly collide).
Peer Pressure

<table>
<thead>
<tr>
<th>Language</th>
<th>C++</th>
<th>Python</th>
<th>Racket</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strategy</td>
<td>Separate</td>
<td>Open</td>
<td>Open</td>
</tr>
<tr>
<td></td>
<td>Chaining</td>
<td>Addressing</td>
<td>Addressing</td>
</tr>
<tr>
<td>Probe</td>
<td>-</td>
<td>Triangular</td>
<td>Double Hash</td>
</tr>
<tr>
<td>Length</td>
<td>Power of 2</td>
<td>Power of 2</td>
<td>Power of 2</td>
</tr>
</tbody>
</table>
Choosing a good Hash

A Hash should be

• Fast – The faster the better.
• Avoid Collisions: Chance of $h(x) = h(y)$ (mod n) is low.
• Open Addressing is more strict
  • Chance of $h(x) = h(y)$ +- small constant is low
• Ideal: Perfectly Uniform
  • Hard to prove, usually checked empirically
Hashing Integers

If the integers are perfectly uniform then the identity function works just fine.

If the integers are actually addresses then they are divisibly by 4 (or 8, on 64 bit systems).

That’s a problem if you used array doubling!
  • Why?
Prime Numbers

(mod p) provides some extra uniformity to poor hash functions. Primes are **needed** for quadratic probe with alternating sign.

It’s fairly costly to “double, then find a prime” on the fly.

So don’t.

primes = [67, 139, 283, 571, 1151, 2311, ...]
To Prime or Not to Prime

There are good reasons **not** to use prime numbers.

Most Hash Tables use powers of two (array doubling).

**Reason**: Modulo by powers of two is very fast. Modulo by primes is not.

e.g. \( (\text{mod } 64) \) means “look at the least significant 6 bits” while \( (\text{mod } 67) \) requires division.
Strings?

A simple hash function simply sums the ASCII / Unicode values of the characters. This has many collisions since all anagrams will have the same sum.

e.g.
\[ h(\text{'ab'}) = h(\text{'ba'}) = 97 + 98 \]
Racket’s String Hash

function string_hash(str):
    h1 = h2 = 0
    for character in str:
        h1 = h1 * 33 + character
        h2 = h2 + character
    return h1,h2

(For double hashing, it’s OK if hash2 is crud, as long as it doesn’t correlate with hash1)
Arbitrary Data

The previous function works for any object, really. Just use

for byte in object’s binary representation:
Perfect Hash Functions

That’s their name…

For any set of n values V, there exists a hash function h(x) that maps all values in V to unique values (modulo some m).

A minimal perfect hash has m == n, meaning you can create a collision free hash table with alpha = 1.0.

You just can only add/remove values from set V.
What’s the time complexity?

Well trivially you can construct a minimal perfect hash function with time complexity $O(n)$. We want $O(1)$. Still possible. Well, the minimal one perhaps not.

It is possible to compute a perfect hash function with $m = n + o(n)$

E.g., it's possible to accomplish $m = 2n-1$
What’s the cost to find $h(x)$ though?

It’s possible to implement dynamic perfect hashing where insert and delete operations have an average case amortized cost of $O(1)$, and lookups have a hard $O(1)$ cost.

(That’s right, it’s not only an average case, but also amortized over $n$ operations!)

Nobody does, but it’s possible!