Hashing

Dov Kruger

Department of Electrical and Computer Engineering
Stevens Institute of Technology

October 10, 2022

Give me feedback! `https://pollev.com/dovkruger`

Hashing is a search technique

Convert the thing you are looking for (the key) into the location to find it using a hash function.

Put the key in bin = hash(key)

Later, when looking to see if the key is there look at bin = hash(key)

It is O(1), independent of the size of the data being searched!

"hello"

$\downarrow$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Simple Uses

Hashing is used to implement

- A set
- A map where each key has a corresponding value.

Later we will learn more complicated uses of hashing to summarize large data

Hashing is Fast, but Complicated

Hash function converts key to location in a table

Unfortunately memory locations are finite

- There are always far more possible values than table entries

Two or more keys can hash to the same location, resulting in a **collision**

- Collisions are unavoidable
- All hash algorithms must include a way to resolve them
- The trick is to keep the number low so that the average speed is high

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Hash Functions

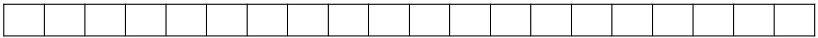
Ideally spread all keys randomly across all locations to minimize collisions

A bad hash function could take all keys and put them in just a single bin

Example: hash(key) $\leftarrow$ 0 (very bad hash function)

insert: "hello", "Fred", "my long string"

$\downarrow$



For hashing words, try summing the letters

Suppose we encode a=1, b=2, ... z =26

```
hash(key)
  sum = 0
  for i = 0 to length(key)
    sum = sum + key[i]
  end
  return sum
end
```

hash("abc") = 1+2+3 = 6

hash("fred") = 6+18+5+4=33

Problem: words with the same letters cat, act, eat, ate, tea, eta

Better Scheme for Strings

To hash strings better, assign different values to the same letters in different positions

Two possible schemes

- Scale by position within the word
- Use base 26 for English letters to form a ânumberâ out of the letters

Example

"hello" = 1*h + 2*e + 3*l
+ 4*l + 5*o
"eat" = 1*e + 2*a + 3*t = 67
"tea" = 1*t + 2*e + 3*a = 31

Base 26 Example:

tea = 20 * 26^2 + 5 * 26 + 1
eat = 5 * 26^2 + 1 * 26 + 20

Problem: Hash Numbers are Huge

Hash numbers typically use the entire 32 or 64 bits of the integer

Compute them modulo the size of your table to keep in range

Example: hash("mystring") = 12578193, table size = 20

select bucket = 12578193 mod 20 = 13

↓

												X							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY

Theory vs. Reality

CLRS (the white book) recommend using tables of size prime number

- Theoretically this is an advantage because there are no beat frequencies
- So, for example use a table size = 32719

In practice, this is not good advice

- A good hash function will distribute values well
- A bad hash function will make table size irrelevant
- Conclusion: It's all about the hash function
- Use table size that is a power of 2 because computing mod is faster

Example n = 32768 2^{15}

To compute hash(key) mod n faster know that: $x\ mod\ 2^k = x\ AND\ (2k - 1)$
So return hash(key) & (n-1)

Great Hash Functions

There is an informal competition between practical programmers doing hashing

Bob Jenkins	https://en.wikipedia.org/wiki/Jenkins_hash_function http://www.burtleburtle.net/bob/hash/doobs.html
Paul Hsieh	http://www.azillionmonkeys.com/qed/hash.html https://gist.github.com/CedricGuillemet/4978020
Fastest Hash?	https://stackoverflow.com/questions/3665247/fastest-hash-for-non-cryptographic-uses
Meta(Facebook)	https://engineering.fb.com/2019/04/25/developer-tools/f14/

A classic probability problem

- With 20 people in the room, what is the probability that at least 2 have the same birthday?
- On the surface, seems small: there are n=365 days and r = 20 days
- About 1/20 of the days are used

Birthday Paradox Solution

$r = 1$ With one person, $p = 0$ of having the same birthday as anyone else

$r = 2$ $p = 1/365$

$r = 3$ $p = 1/365 + 2/365$ (the 3rd person has 2 chances)

$r = 4$ $p = 1/365 + 2/365 + 3/365$

.

.

.

$r = 20$ $p = 1/365 + 2/365 + ... + 19/365 = 0.52$

Implications of Birthday Paradox

The probability of zero collisions is almost zero
Therefore, you will have to figure out a way to resolve collisions
No hash function, no matter how good, will eliminate all collisions!

On the other hand, you have to have a good hash function **No collision scheme, no matter how good, will make up for a bad hash function!**

We want a hash function that distributes (on average) 1 value to each bin
Some bins will be empty
Some bins will have 1 element
Only a few will have more
The average lookup time will be O(1)

There are three ways of building a hash table
Each one uses a different method to handle the inevitable collision problem

- Linear Probing
- Linear Chaining
- Perfect Hashing

In addition, quadratic chaining is a minor tweak to linear chaining, and not very useful

You can read about it in the notes for completeness

Linear Probing

Calculate the position using a hash function

If the position is already full, use the next one to the right

If at the end of the table, start at the beginning again

Example: n=20, hash(key) = key mod n, insert 33, 13, 73, 20, 14

													33						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

STEVENS INSTITUTE of TECHNOLOGY THE INNOVATION UNIVERSITY®

Linear Probing

Calculate the position using a hash function

If the position is already full, use the next one to the right

If at the end of the table, start at the beginning again

Example: n=20, hash(key) = key mod n, insert 33, 13, 73, 20, 14

collision, so insert to the right

													33	13					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Linear Probing

Calculate the position using a hash function

If the position is already full, use the next one to the right

If at the end of the table, start at the beginning again

Example: n=20, hash(key) = key mod n, insert 33, 13, 73, 20, 14

collision in both 13 and 14, so insert in bin 15

↓

													33	13	73				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Linear Probing

Calculate the position using a hash function

If the position is already full, use the next one to the right

If at the end of the table, start at the beginning again

Example: n=20, hash(key) = key mod n, insert 33, 13, 73, 20, 14

↓

20													33	13	73				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Calculate the position using a hash function

If the position is already full, use the next one to the right

If at the end of the table, start at the beginning again

Example: n=20, hash(key) = key mod n, insert 33, 13, 73, 20, 14

collisions in bin 14, 15, insert in 16

↓

20													33	13	73	14			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

If at the end of the table, then the next position to the right is the beginning. Example: n=20, hash(key) = key mod n, insert 19, 39

collisions in bin 14, 15, insert in 16

$\downarrow$

																			19
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Linear Probing: Wrapping Around

If at the end of the table, then the next position to the right is the beginning. Example: n=20, hash(key) = key mod n, insert 19, 39

collisions in bin 19, insert in 0

↓

39																			19
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Linear Probing: Collision Avoidance Requires Empty Bins

In Linear Probing, the previous example shows that collisions are caused by

- Two values having the same hash value (example 13)
- Neighboring bins when one of them spills over

Because of the fact that one bin can "ruin the neighborhood" we need to make sure there are as many empty bins as possible

To avoid collisions, use 100% extra bins
or, 50% of the bins should be empty

- For n symbols, make sure there are 2n bins
- The table can be made bigger and performance improves, diminishing returns
- If the table is made smaller collisions will rapidly increase
- For a table of n bins, trying to insert n+1 symbols is an infinite loop!

STEVENS INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

Pros and Cons of Linear Probing

Pros

- Single contiguous block of memory is fast, simple to allocate

Cons

- One bin with many collisions can ruin the neighborhood for other bins
- A lot of empty bins are required (space overhead, O(n))
- If values are stored in the table, one value must be reserved to show empty
  - The un-value

Here, we are using zero as the "un-value"

Cannot put the number zero in the table

0	0	0	0	0	21	0	0	0	0	0	0	0	0	0	0

Implementing a Hash Map Using Linear Probing

A **map** is a data structure allowing looking up a **key** to find a corresponding **value**

Often, hashing is used to find the (key,value) pair because it is the fastest

In this case, create a structure holding a key and value and put in each bin

Unfortunately, keeping them inline means that all the empty bins are wasted space

key	key	key	key	key	key	key	key
value	value	value	value	value	value	value	value

0	0	0	0	4	0	0	0
wasted	space	wasted	space	hello	space	wasted	space

Implementing empty bins using pointers

In order to solve this problem a pointer can be used

- Wastes less space
- All key values can be represented
- Slower and requires many separate memory allocations

null	null	null	null	null		null	null		null	null	null	null	null	null	null
					21			40							

Test Yourself

Given Hash function f(key) = (sum the letters) mod n
Letters use real ASCII values: https://en.wikipedia.org/wiki/ASCII
n=16
Example: hash("abc") = 97 + 98 + 99 = 294 mod 16 = 9
insert words "hello", "eat", "tea", "Vlad the Impaler" into the table

5.3 Linear Chaining

- Second Approach to Collision Resolution
- Each bin is a linked list (head pointer only)
- Empty bins just have a null pointer
- Empty bins can be smaller than linear probing
- Not a single chunk of linear memory, but advantage: each bin is independent

Linear Chaining: Collision Resolution

For linear chaining, if there are collisions then the linked list in one bin gets too long

- Best solution is big enough table
- Use minimum 25-30% extra space

Building a Hash Map with Linear Chaining

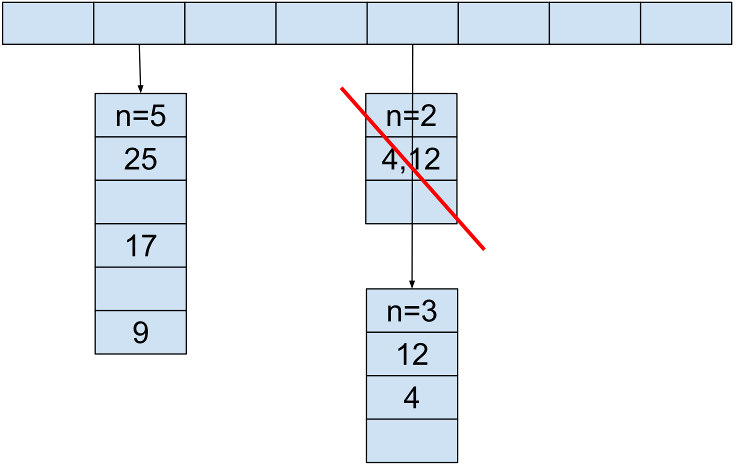
Instrumenting your HashMap

Instrumenting your HashMap

Perfect Hashing

Perfect Hashing is a technique for creating a hash table without collisions The birthday paradox says this is impossible. So how does it work?

- Works only on a fixed data set. Plenty of time to figure it out once
- Use a two-level table. The first level has some collisions
- Within each bin is a second table resolving the collisions
- Perfect hashing is a hash table of hash tables


n=5
25
17
9
n=2
4,12
n=3
12
4

5.x Applications of Hashing

There are many uses of hashing

- Implementing a set
- Implementing a map of keys to values
- Summarizing a large block of bytes as a single number

In addition, there are cryptographically secure hashes with very interesting properties.