Hashing

David Croft

Coventry University

david.croft@coventry.ac.uk

2016

Overview

Lookup

Already seen binary search faster than linear.

- But binary search only works on ordered sequences.
  - Sorted list/array, Binary Search Trees (BSTs) etc.
- $O(\log n)$ vs. $O(n)$.
- What's better than $O(n)$?
  - $O(1)$.

Can we lookup values in $O(1)$ time?

Unordered sequence

C

Want to store information on all UK motorways.

- 49 motorways.

Option 1.

- Unordered sequence.
- List/array/vector.
- Finding specific motorway is $O(n)$.
- Space required, $O(n)$.

Pos	Motorways
0	M9
1	M55
2	M898
3	M4
4	M1
5	M6
...	
45	M2
46	M56
47	M53
48	M3

Ordered sequence

C

Option 2.

- Ordered sequence.
- Sorted List/array/vector or BST.
- Finding specific motorway is $O(\log n)$.
- Space required, $O(n)$.

Pos	Motorways
0	M1
1	M2
2	M3
3	M4
4	M5
5	M6
	...
45	M606
46	M621
47	M876
48	M898

Lookup table

C

Option 3

- Lookup table.
- Each motorway stored in position corresponding to it's number.
  - E.g. M1 in position 1, M53 in position 53.
- Finding specific motorway is $O(1)$.
  - Very fast.
- Space required, $O(max(n))$, 899 spaces.
  - Very inefficient...in this case.
  - Can be VERY efficient, massive time savings for small memory cost.

Pos	Motorways
0	
1	M1
2	M2
3	M3
...	...
53	M53
54	M54
55	M55
56	M56
...	...
894	
895	
896	
897	
898	M898

Hash table

C

Option 4

- Hash table.
- Pass each motorway through a hash function.
- Store hashes in lookup table.
- Finding specific motorway is $O(1)$ ish.
- Space required, $O(n)$.

Hash table

C

Hash tables.
Unordered associative arrays.

- Unordered - we have no control over the item orders.
- Associative - Lookup a value based on a key.
  - I.e. Python `dict()`, C++ `map<>`.
- Fast.
  - $O(1)$ lookup (potentially).
  - $O(1) \leq$ Reality $\leq O(n)$.

What is a hash?

C

Hash is just a number.

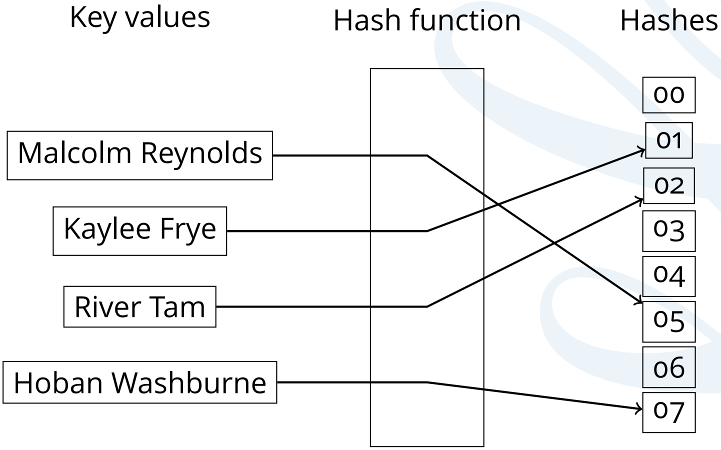
- Based on some other value.
- Motorway example hash could just be the M-number.
  - I.e. M898 $\rightarrow$ 898.

What if key is not an int?

- Hash function.
- Converts an input of any size/range to a fixed size/range.
- Related to, but distinct from:
  - Checksums.
  - Fingerprints.
  - Parity codes.

What is hash? II

C

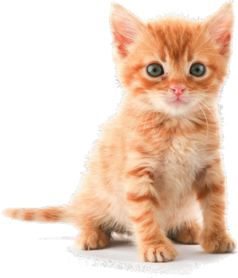


Hashes

I

Signs of a good hash table hashing algorithms.

- Computationally lightweight.
- Evenly distributed hashes.
  - E.g. `len()` would be terrible hash function, loads of different inputs produce same value.

Input	CRC32 hash
`"Small text."`	3840495446
`1`	3523407757
`[1,2,3,4,5]`	1191942644
"On the Origin of Species" 158,454 words	3877468994
	192271774

Hash function

A

So how does a hash function work?

- Depends on hash function and purpose.
- Not going to be implementing any real algorithms.
  - Optimized to be very fast, not understandable.
  - Generally full of binary representations.
  - Bit shifting.

- Simple hashing algorithm (division method).
  1. Break the thing being hashed into blocks.
     - 1, 2, 4, 8, 16 bytes in size.
  2. Add up all the blocks.
  3. Modulo by a prime number.

Cryptographic hashses.

A

Side note, different hashes for different purposes.

- Hash table hashes.
  - Computationally lightweight as possible.
- Cryptographic hashes.
  - Computationally lightweight-ish to go key $\rightarrow$ hash.
  - Computationally expensive to go hash $\rightarrow$ key.
  - MD (Message-Digest algorithm)
    - Famously MD5, widely used, no longer secure.
    - MD6 still good.
  - SHA (Secure Hash Algorithm)
    - SHA-0, SHA-1 not secure.
    - SHA-2, SHA-3 still good.

Wait a second

I

Hang on a minute, there's problem.

- Hashed "buckeroo" with CRC32.
  - CRC32 **NOT** best choice for hash table but is easy and widespread.
  - Hash of 1306201125
- So our hash table needs at least 1,306,201,125 slots.
  - 4 bytes per integer * 1306201125 slots = 5.2 gigabytes.
  - Not going to work.
- Solution? Take the modulo of the hash.
  - Create table of small size.
  - `slot = hash % len(hashtable)`

Collisions

C

Converting big sequences into short hashes.

- Any downsides?
- Some distinct sequences **MUST** produce same hash.
- Hash collision.

- I.e. Hashing an `int`
  - `int` has 4 billion possible values.
  - If hash is one byte, then 256 possible slots.
  - 4 billion possible values in 256 possible slots, collisions will happen.

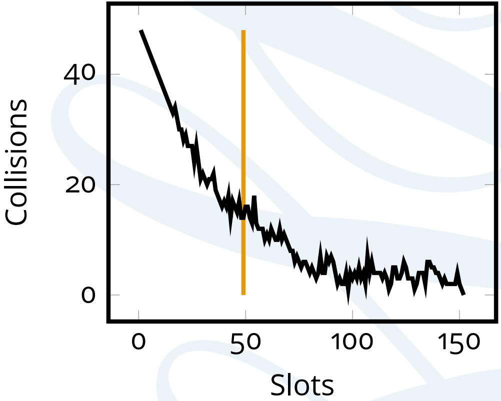
- E.g. CRC32 hash.
  - "plumless" $\rightarrow$ 1306201125 $\leftarrow$ "buckeroo"

Table size

I

Bigger the hash table == less collisions.

- Ideal motorway hash table.
  - 49 motorways.
  - 49 slots.
  - 0 collisions.
- Reality.
  - 49 slots.
  - 14 collisions.
  - I.e. M18, M67 and M606 in slot 18.



- Optimal.
  - 152 slots.
  - 0 collisions.
  - Size only 3.1 times ideal.

Load factor

Number of collisions depends on load factor (l).

- Ratio of elements (n) to available slots (k).
  - $l = \frac{n}{k}$
- High load = lots of collisions (probably).
- Low load = few collisions (probably).
- $l > 1.0$ = definitely some collisions.

- Previous motorway example.
  - 49 motorways, 49 slots, $l = 1$
  - 49 motorways, 152 slots, $l = 0.31$

Adjusting for load

A

Real world hash tables.

- Automatically resize to provide more slots as load increases.

- Advantages
  - Table size adjusts for the amount of elements stored in it.
  - Minimal wasted memory.

- Disadvantages
  - Have to shuffle everything around when the table resizes.
  - Not all that time consuming.

Handle collisions

I

So what do we do when we have collisions?

- If have ≥ 1 elements then collisions are possible.
  - Regardless of table size.
- Two main approaches

Separate chaining.

- Each slot is a linked list.
- Infinitely resizeable.
- Add new item to end of the list.

Open addressing.

- Slot is already full?
- Try next slot until an empty one is found.

Handle collisions II

I

Separate chaining.

buckeroo	1306201125

$\Rightarrow$

Slot				
0	$\rightarrow$	limpet	$\rightarrow$	
1	$\rightarrow$	zombie	$\rightarrow$	
2	$\rightarrow$			
3	$\rightarrow$	plumless	$\rightarrow$	buckeroo
4	$\rightarrow$			
5	$\rightarrow$	gondola	$\rightarrow$	

Open addressing

buckeroo	1306201125

$\Rightarrow$

Slot	
0	limpet
1	zombie
2	
3	plumless
4	buckeroo
5	gondala

Hash tables in Python

C

Dictionaries!

```
import sys

def main():
    motorways = {}

    motorways["M1"] = (193.5, 1959)
    motorways["M2"] = (25.7, 1963)
    motorways["M3"] = (58.6, 1971)
    motorways["M4"] = (191.9, 1961)
    #                     [...]
    motorways["M898"] = (0.5, 1985)

    print( 'The %s is %0.1f miles long' % ("M4",
    ↪   motorways["M4"][0]) )
    print( 'The %s opened in %d' % ("M898",
    ↪   motorways["M898"][1]) )

if __name__ == '__main__':
    sys.exit(main())
```

Hash tables in C++

C

Maps!

```
#include <iostream>
#include <map>
using namespace std;

int main()
{
    map< string, pair<float,int> > motorways;

    motorways.emplace( "M1", make_pair<float,int>(193.5, 1959) );
    //                  [...]
    motorways.emplace( "M898", make_pair<float,int>( 0.5, 1985 ) );

    cout << "The " << "M1" << " is " <<
        motorways.find("M1")->second.first << " miles long" << endl;

    cout << "The " << "M898" << " opened in " <<
        motorways.find("M898")->second.second << endl;

    return 0;
}
```

A

Hashes have many other possible applications.

- Finding duplicates.
  - Hash table but count number of things in each slots.
- Similarity comparisons.
  - E.g. Soundex, Metaphone.
  - Names that sound the same have same hash.
- Image recognition.
- Bloom filters.
  - Are almost magic.

Harry Potter and the Bloom Filter

Bloom Filters

A

Neat trick with hashes.

- Can 'store' 1000 things in the space for 100.
  - Doesn't actually store the items.
  - Can say if an element is not a member of a set.
  - Can say if element is probably a member of a set

How?

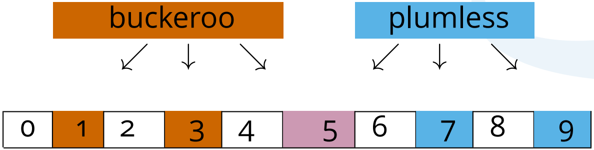
A

To add a value to the filter.

- Hash the value using multiple different functions.
- Mark the slots for each of those hashes.

To test for a value in the filter.

- Hash the value using all the functions.
  - If not all the slots are marked then value not in filter.
  - If all slots are marked then value *probably* in filter.



Why?

A

Bloom filters are almost magic.

- 1000 items
- $\leq$ 1% error
- 7 hash functions
- 9586 slots
  - 1 bit per slot.
- 'Store' 1000 integers in the space for 300.
  - Bigger variables mean bigger savings.

Quiz

Recap

The end of 122COM...
...or is it?