Universal Hashing

https://cs.pomona.edu/classes/cs140/

Hash Tables

Operations:

- Insert
- Delete
- Look-up

O(1)

What are they not good for?

Guaranteed constant running time for those operations if:

1. If the hash table is properly implemented, and
2. The data is non-pathological.

Hash Table Load

$$\alpha := \frac{\#\ of\ objects\ in\ the\ hash\ table}{\#\ of\ buckets}$$

- What is the maximum possible α for <u>separate chaining</u>?
- What is the maximum possible α for <u>open addressing</u>?

Hash Table Load

$$\alpha := \frac{\#\ of\ objects\ in\ the\ hash\ table}{\#\ of\ buckets}$$

1. $\alpha = O(1)$ is necessary to ensure that hash table operations happen in constant time
2. For open addressing, you typically need $\alpha \ll 1$ 0.75 is rule of thumb

- Thus, for good hash table performance you must control the load
- How do you control the load?

Pathological Data Sets

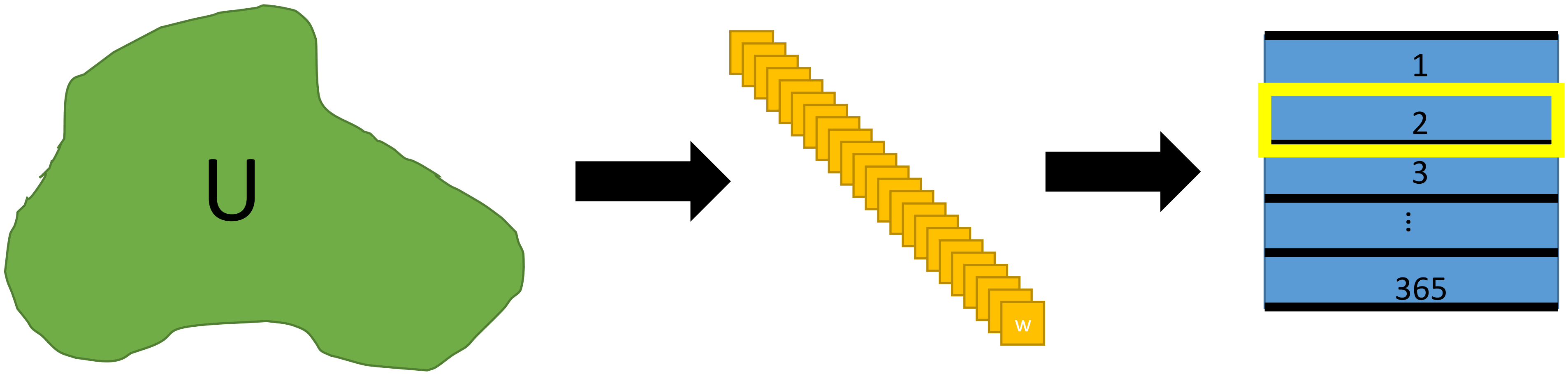
- We want our hash functions to “spread-out” the data (i.e., minimize collisions)
- Unfortunately, no perfect hash function exists (it’s impossible)
- You can create a pathological data set for any hash function

Pathological Data Sets

Purposefully select only the elements that map to the same bucket.

Fix (set) the hash function h(x) → {0, 1, …, n-1}, where n is the number of buckets in the hash table and n << |U|

With the pigeonhole principle, there must exist a bucket i, such that at least |U|/n elements of U hash to i under h



Pathological Data Set Example

- We want to store student student ID numbers in a hash table.
- We will store about 30 students worth of data
- Let’s use a hash table with 87 buckets
- Let’s use the final three numbers as the hash

```python
s = 30
n = 87

def hash_fcn(id_number):
    return id_number % n

id_numbers = [randint(1000000, 9999999) for _ in range(s)]
hash_values = map(hash_fcn, id_numbers)
print('Number of unique student IDs:', len(set(id_numbers)))
print('Number of unique hash values:', len(set(hash_values)))

id_numbers_pathological = [round(num,  -2) for num in id_numbers]
hash_values_pathological = map(hash_fcn, id_numbers_pathological)
print('Number of unique student IDs:', len(set(id_numbers_pathological)))
print('Number of unique hash values:', len(set(hash_values_pathological)))
```

Output:

```
Number of unique student IDs: 30
Number of unique hash values: 28

Number of unique student IDs: 30
Number of unique hash values: 1
```

Real World Pathological Data

- Denial of service attack
- A study in 2003 found that they could interrupt the service of any server with the following attributes:

1. The server used an open-source hash table
2. The hash table uses an easy-to-reverse-engineer hash function

- How does reverse engineering the hash function help an attacker?

Solutions to Pathological Data

Use a cryptographic hash function

- Infeasible to create pathological data for such a function (but not theoretically impossible)

Use randomization (Can still be an open-source implementation!)

1. Create a family of hash functions
2. Randomly pick one at runtime

Universal Hashing

Let H be a **set** of hash functions mapping U to {0, 1, ..., n-1}

The family H is <u>universal</u> if and only if for all x, y in U

$\Pr(h(x) = h(y)) \leq 1/n$

Probability of a collision

where h is chosen uniformly at random from H

Basically, the hash functions don't all have the same flaw where they map a set of inputs to the same bucket.

Example: Hashing IP Addresses

$|U| = 2^{32} = 256^4$

=

4,294,967,296

- What is U? And how big is U?
- U includes all IP addresses, which we'll denote as 4-tuples
  example: $X = (x_1, x_2, x_3, x_4)$ where x_i is in [0, 255]
- Let n = some prime number that is near a multiple of the number of objects we expect to store
  example: $|S| = 500$, we set $n = 997$
- Let H be our set of hash functions
  example: $h(x) = A \text{ dot } X \bmod n = (a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4) \bmod n$
  where $A = (a_1, a_2, a_3, a_4)$ and a_i is in [0, n-1]
  H includes all combinations the coefficients in A

$|H| = n^4$

=

988 billion

```python
n = 997

def ip_hash_fcn(X, A):
    return sum([x * a for x, a in zip(X, A)]) % n

ip_address = [randrange(256) for _ in range(4)] # i.e., 192.168.3.7
hash_coeff = [randrange(n) for _ in range(4)]

print("IP address          :", ".".join(map(str, ip_address)))
print("Hash coefficients :", hash_coeff)
print("Hash value          :", ip_hash_fcn(ip_address, hash_coeff))
```

x_1 x_2 x_3 x_4

```
IP address        : 227.75.113.191
```

a_1 a_2 a_3 a_4

```
Hash coefficients : [394, 429, 328, 78]

Hash value        : 97
```

Example: Hashing IP Addresses

Theorem: the family H is universal

$$\frac{\#\ of\ functions\ that\ map\ x\ and\ y\ to\ the\ same\ location}{total\ \#\ of\ functions} \leq \frac{1}{n}$$

- Let H be a **set** of hash functions mapping U to {0, 1, …, n-1}
- The family H is universal if and only if for all x, y in U
- `Pr(h(x) = h(y)) ≤ 1/n`
- where h is chosen uniformly at random from H

Hashing IP Addresses Proof

- Consider two distinct IP addresses X and Y
- Assume that $x_4 \neq y_4$ (they might differ in all parts)
  - The same argument will hold regardless of which part of the tuple we consider
- Based on our choice of h_i, what is the probability of a collision?
  - Or what fraction of h_is cause a collision? `Pr[h(X) = h(Y)]`
- Where h_i is any of the hash function from H

- We want to show that ≤ 1/n of the billions of hash functions have a collision for X and Y

Theorem: for any possible hash function, the probability of a collision between objects X and Y is $\leq \frac{1}{n}$

Hash functions are selected from the hash family by randomly generating four values for A

Collision between objects X and Y

$$h(X) = h(Y)$$

$$(A \cdot X)\ mod\ n = (A \cdot Y)\ mod\ n$$

$$(a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4)\ mod\ n = (a_1y_1 + a_2y_2 + a_3y_3 + a_4y_4)\ mod\ n$$

$$0 = a_1(y_1 - x_1) + a_2(y_2 - x_2) + a_3(y_3 - x_3) + a_4(y_4 - x_4)\ mod\ n$$

Theorem: for any possible hash function, the probability of a collision between objects X and Y is $\leq \frac{1}{n}$

Hash functions are selected from the hash family by randomly generating four values for A

$$0 = a_1(y_1 - x_1) + a_2(y_2 - x_2) + a_3(y_3 - x_3) + a_4(y_4 - x_4) \; mod \; n$$

Something must be different between X and Y. Let's assume that $x_4 \neq y_4$

$$a_4(x_4 - y_4) \; mod \; n = a_1(y_1 - x_1) + a_2(y_2 - x_2) + a_3(y_3 - x_3) \; mod \; n$$

Fixed, non-zero value

Assume n is prime.

From here we are going to **fix** our choices of a_1, a_2, and a_3 and let a_4 be a random variable

Principle of Deferred Decisions

We want to show that for any value of a_4 we have a $\frac{1}{n}$ chance of a collision.

Theorem: for any possible hash function, the probability of a collision between objects X and Y is $\leq \frac{1}{n}$

Something must be different between X and Y. Let's assume that $x_4 \neq y_4$

Fixed, non-zero value

Assume n is prime.

$$a_4(x_4 - y_4) \bmod n = a_1(y_1 - x_1) + a_2(y_2 - x_2) + a_3(y_3 - x_3) \bmod n$$

From here we are going to **fix** our choices of a_1, a_2, and a_3 and let a_4 be a random variable

Principle of Deferred Decisions

We want to show that for any value of a_4 we have a $\frac{1}{n}$ chance of a collision.

How many choices of a_4 satisfy the above equation?

- Our RHS is fixed! It is just some number in $[0, n\text{-}1]$ because X, Y, and a_1, a_2, a_3 are fixed
- If n is a prime number, then the LHS is equally likely to be any number from $[0, n\text{-}1]$
  - This claim requires some number theory to properly prove

Unique multiplicative

Thus, based on our choice for a_4, we have that $Pr(h(X) = h(Y)) = 1/n$

Prime number for n

$n = 7, x_4 = 3, y_4 = 1$

a_4	$a_4(x_4 - y_4) \ mod \ n$
0	0
1	2
2	4
3	6
4	1
5	3
6	5

X = (x1, x2, x3, x4) where xi is in [0, 255]

A = (a1, a2, a3, a4) and ai is in [0, n-1]

|S| = 500

n = 997

h(x) = $(A \cdot X) \ mod \ n$

And H includes all combinations for the coefficients in A

What do we want in the second column?

Prime number for n

n = 7, x_4 = 3, y_4 = 1

a_4	$a_4(x_4 - y_4) \bmod n$
0	0
1	2
2	4
3	6
4	1
5	3
6	5

n = 7, x_4 = 4, y_4 = 1

a_4	$a_4(x_4 - y_4) \bmod n$
0	0
1	3
2	6
3	2
4	5
5	1
6	4

Non-Prime number for n

x4-y4 shares factors with n

$n = 8, x_4 = 3, y_4 = 1$

a_4	$a_4(x_4 - y_4) \bmod n$
0	0
1	2
2	4
3	6
4	0
5	2
6	4
7	6

$n = 8, x_4 = 4, y_4 = 1$

a_4	$a_4(x_4 - y_4) \bmod n$
0	0
1	3
2	6
3	1
4	4
5	7
6	2
7	5

Summary

- We cannot create a hash function that prevents creation of a pathological dataset

- As long as the hash function is known, a pathological dataset can be created

- We can create families of hash functions that make it infeasible to guess which hash function is in use