

# Hashing (Application of Probability)

Ashwinee Panda

Final CS 70 Lecture!

9 Aug 2018

# Overview

- ▶ Intro to Hashing
- ▶ Hashing with Chaining
- ▶ Hashing Performance
- ▶ Hash Families
- ▶ Balls and Bins
- ▶ Load Balancing
- ▶ Universal Hashing
- ▶ Perfect Hashing

## What's the point?

Although the name of the class is “Discrete Mathematics and Probability Theory”, what you've learned is not just theoretical but has far-reaching applications across multiple fields. Today we'll dive deep into one such application: hashing.

# Intro to Hashing

What's hashing?

- ▶ Distribute key/value pairs across bins with a *hash function*, which maps elements from large universe  $\mathbb{U}$  (of size  $n$ ) to a small set  $\{0, \dots, k - 1\}$
- ▶ Given a key, always returns one integer
- ▶ Hashing the same key returns the same integer;  $h(x) = h(x)$
- ▶ Hashing two different keys might not always return different integers
- ▶ Collisions occur when  $h(x) = h(y)$  for  $x \neq y$

You may have heard of SHA256, a special class of hash function known as a cryptographic hash function.

## Hashing with Chaining

In CS 61B you learned one particular use for hashing: hash tables with linked lists.

Pseudocode for hashing one key with a given hash function:

```
def hash_function(x):  
    return x mod 7  
hash = hash_function(key)  
linked_list = hash_table[hash]  
linked_list.append(key)
```

- ▶ Mapping many keys to the same index causes a COLLISION
- ▶ Resolve collisions with “chaining”
- ▶ Chaining isn't perfect; we have to search through the list in  $O(\ell)$  time where  $\ell$  is the length of the linked list
- ▶ Longer lists mean worse performance
- ▶ Try to minimize collisions

## Hashing Performance

Operation	Average-Case	Worst-Case
Search	$O(1)$	$O(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

- ▶ Hashing has great average-case performance, poor worst-case
- ▶ Worst-case is when all keys map to the same bin (collisions); performance scales as maximum number of keys in a bin

An adversary can induce the worst case (adversarial attack)

- ▶ For  $h(x) = x \bmod 7$ , suppose our set of keys is all multiples of 7!
- ▶ Each item will hash to the same bin
- ▶ To do any operation, we'll have to go through the entire linked list

## Hash Families

- ▶ If  $|\mathbb{U}| \geq (n - 1)k + 1$  then the Pigeonhole Principle says one bucket of the hash function must contain at least  $n$  items
- ▶ For any hash function, we might have keys that all map to the same bin—then our hash table will have terrible performance!
- ▶ Seems hard to pick just one hash function to avoid worst-case
- ▶ Instead, develop randomized algorithm!
- ▶ Randomized algorithms use randomness to make decisions
- ▶ Quicksort expects to find the right answer in  $O(n \log n)$  time but may run for  $O(n^2)$  time (CS 61B)
- ▶ We can restart a randomized algorithm as many times as we wish, to make the  $\mathbb{P}[\text{fail}]$  arbitrarily low
- ▶ To guard against an adversary we generate a hash function  $h$  uniformly at random from a hash family  $\mathcal{H}$
- ▶ Even if the keys are chosen by an adversary, no adversary can choose bad keys for the entire family simultaneously, so our scheme will work with high probability

## Balls and Bins

- ▶ If we want to be REALLY random, we'd see hashing as just balls and bins
- ▶ Specifically, suppose that the random variables  $h(x)$  as  $x$  ranges over  $\mathbb{U}$  are independent
- ▶ Balls will be the keys to be stored
- ▶ Bins will be the  $k$  locations in hash table
- ▶ The hash function maps each key to a uniformly random location
- ▶ Each key (ball) chooses a bin uniformly and independently
- ▶ How likely can collisions be? The probability that two balls fall into same bin is  $\frac{1}{k^2}$
- ▶ Birthday Paradox: 23 balls and 365 bins  $\implies$  50% chance of collision!
- ▶  $n \geq \sqrt{k} \implies \frac{1}{2}$  chance of collision

## Balls and Bins

$X_i$  is the indicator random variable which turns on if the  $i^{\text{th}}$  ball falls into bin 1 and  $X$  is the number of balls that fall into bin 1

- ▶  $\mathbb{E}[X_i] = \mathbb{P}[X_i = 1] = \frac{1}{k}$
- ▶  $\mathbb{E}[X] = \frac{n}{k}$

$E_i$  is the indicator variable that bin  $i$  is empty

- ▶ Using the complement of  $X_i$  we find  $\mathbb{P}[E_i] = (1 - \frac{1}{k})^n$

$E$  is the number of empty locations

- ▶  $\mathbb{E}[E] = k(1 - \frac{1}{k})^n$
- ▶  $k = n \implies \mathbb{E}[E] = n(1 - \frac{1}{n})^n \approx \frac{n}{e}$  and  $\mathbb{E}[X] = \frac{n}{n}$
- ▶ How can we expect 1 item per location (very intuitive with  $n$  balls and  $n$  bins) and also expect more than a third of locations to be empty?

$C$  is the number of bins with  $\geq 2$  balls

- ▶  $\mathbb{E}[C] = n - k + \mathbb{E}[E] = n - k + k(1 - \frac{1}{k})^n$

# Load Balancing

- ▶ Distributed computing: evenly distribute a workload
- ▶  $m$  identical jobs,  $n$  identical processors (may not be identical but that won't actually matter)
- ▶ Ideally we should distribute these perfectly evenly so each processor gets  $\frac{m}{n}$  jobs
- ▶ Centralized systems are capable of this, but centralized systems require a server to exert a degree of control that is often impractical
- ▶ This is actually similar to balls and bins!
- ▶ Let's continue using our random algorithm of hashing
- ▶ Let's try to derive an upper bound for the maximum length, assuming  $m = n$

# Load Balancing

$H_{i,t}$  is the event that  $t$  keys hash to bin  $i$

- ▶  $\mathbb{P}[H_{i,t}] = \binom{n}{t} \left(\frac{1}{n}\right)^t \left(1 - \frac{1}{n}\right)^{n-t}$
- ▶ Approximation:  $\binom{n}{t} \leq \frac{n^n}{t^t (n-t)^{n-t}}$  by Stirling's formula
- ▶ Approximation:  $\forall x > 0, \left(1 + \frac{1}{x}\right)^x \leq e$  by the limit
- ▶ Because  $\left(1 - \frac{1}{n}\right)^{n-t} \leq 1$  and  $\left(\frac{1}{n}\right)^t = \frac{1}{n^t}$  we can simplify
- ▶ 
$$\binom{n}{t} \left(\frac{1}{n}\right)^t \left(1 - \frac{1}{n}\right)^{n-t} \leq \frac{n^n}{t^t (n-t)^{n-t} n^t} = \frac{n^{n-t}}{t^t (n-t)^{n-t}}$$
$$= \frac{1}{t^t} \left(1 + \frac{t}{n-t}\right)^{n-t} = \frac{1}{t^t} \left(\left(1 + \frac{t}{n-t}\right)^{\frac{n-t}{t}}\right)^t \leq \frac{e^t}{t^t}$$

$M_t$ : event that max list length hashing  $n$  items to  $n$  bins is  $t$

$M_{i,t}$ : event that max list length is  $t$ , and this list is in bin  $i$

- ▶  $\mathbb{P}[M_t] = \mathbb{P}[\bigcup_{i=1}^n M_{i,t}] \leq \sum_{i=1}^n \mathbb{P}[M_{i,t}] \leq \sum_{i=1}^n \mathbb{P}[H_{i,t}]$
- ▶ Identically distributed loads means  $\sum_{i=1}^n \mathbb{P}[H_{i,t}] = n\mathbb{P}[H_{i,t}]$

The probability that the max list length is  $t$  is at most  $n\left(\frac{e}{t}\right)^t$

# Load Balancing

Expected max load is  $\sum_{t=1}^n t\mathbb{P}[M_t]$  where  $\mathbb{P}[M_t] \leq n(\frac{e}{t})^t$

- ▶ Split sum into two parts and bound each part separately.
- ▶  $\beta = \lceil \frac{5 \ln n}{\ln \ln n} \rceil$ . How did we get this? Take a look at Note 15.
- ▶  $\sum_{t=1}^n t\mathbb{P}[M_t] = \sum_{t=1}^{\beta} t\mathbb{P}[M_t] + \sum_{t=\beta}^n t\mathbb{P}[M_t]$

Sum over smaller values:

- ▶ Replace  $t$  with the upper bound of  $\beta$
- ▶  $\sum_{t=1}^{\beta} t\mathbb{P}[M_t] \leq \sum_{t=1}^{\beta} \beta\mathbb{P}[M_t] = \beta \sum_{t=1}^{\beta} \mathbb{P}[M_t] \leq \beta$   
as the sum of disjoint probabilities is bounded by 1

Sum over larger values:

- ▶ Use our expression for  $\mathbb{P}[H_{i,t}]$  and see that  $\mathbb{P}[M_t] \leq \frac{1}{n^2}$ .
- ▶ Since this bound decreases as  $t$  grows, and  $t \leq n$ :
- ▶  $\sum_{t=\beta}^n t\mathbb{P}[M_t] \leq \sum_{t=\beta}^n n\frac{1}{n^2} \leq \sum_{t=\beta}^n \frac{1}{n} \leq 1$
- ▶ Expected max load is  $O(\beta) = O(\frac{\ln n}{\ln \ln n})$

# Universal Hashing

What we've been working with so far is “ $k$ -wise independent” hashing or fully independent hashing.

- ▶ For any number of balls  $k$ , the probability that they fall into the same bin of  $n$  bins is  $\frac{1}{n^k}$
- ▶ Very strong requirement!
- ▶ Fully independent hash functions require a large number of bits to store

Do we compromise, and make our worst case worse so we can have more space?

- ▶ Often you do have to sacrifice time for space, vice-versa
- ▶ But not this time! Let's inspect our worst-case
- ▶ Collisions only care about two balls colliding

We don't need “ $k$ -wise independence” we only need “2-wise independence”

# Universal Hashing

## Definition of Universal Hashing

- ▶ We say  $\mathcal{H}$  is **2-universal** if  $\forall x \neq y \in \mathbb{U}, \mathbb{P}[h(x) = h(y)] \leq \frac{1}{k}$
- ▶ Let  $\mathcal{C}_x$  be the number of collisions with item  $x$ , and  $\mathcal{C}_{x,y}$  be the indicator that items  $x$  and  $y$  collide
- ▶ This implies  $\mathbb{E}[\mathcal{C}_x] = \sum_{y \in \mathbb{U} \setminus \{x\}} \mathbb{E}[\mathcal{C}_{x,y}] \leq \frac{n}{k} = \alpha$
- ▶  $\alpha$  is called the “load factor”

If we can construct such an  $\mathcal{H}$  then we'll expect constant-time operations. . . pretty cool!

# Universal Hashing

## Defining hashing scheme

- ▶ Our universe has size  $n$  and our hash table has size  $k$
- ▶ Say  $k$  is prime and  $n = k^r$ .  $\forall x \in \mathbb{U} : x = [x_1 \ x_2 \ \cdots \ x_r]$
- ▶ Represent our key as a vector  $[x_1 \ x_2 \ \cdots \ x_r]$  s.t. for all  $i$ ,  $x_i \in \{0, \dots, k-1\}$
- ▶ Choose  $n$ -length random vector  $V = [v_1 \ v_2 \ \cdots \ v_r]$  from  $\{0, \dots, k-1\}^r$  and take dot product

## Proving universality

- ▶  $x \neq y \implies \exists i : x_i \neq y_i$  (at least one index different)
- ▶  $\mathbb{P}[h(x) = h(y)] = \mathbb{P}[\sum_{i=1}^r v_i x_i = \sum_{i=1}^r v_i y_i]$   
 $= \mathbb{P}[v_i(x_i - y_i) = \sum_{j \neq i} v_j y_j - \sum_{j \neq i} v_j x_j]$
- ▶  $x_i - y_i$  has an inverse modulo  $k$
- ▶  $\mathbb{P}[v_i = \frac{\sum_{j \neq i} v_j y_j - \sum_{j \neq i} v_j x_j}{x_i - y_i}] = \frac{1}{k}$

There are lots of universal hash families; this is just one!

# Static Hashing

The dictionary problem (static):

- ▶ Store a set of items, each is a (key, value) pair
- ▶ The number of items we store will be roughly the same size as the hash table (i.e., we want to store  $\approx k$  items)
- ▶ Support only one operation: search
- ▶ Binary search trees: search typically takes  $O(\log k)$  time
- ▶ Hash table: search takes  $O(1)$  time
- ▶ Distinct from the dynamic dictionary problem

## Perfect Hashing for Static Dictionaries

$h$  is perfect for a given set of keys if all lookups are  $O(1)$

- ▶ Hash into table  $A$  of size  $k$  with universal hashing
- ▶ We'll end up with some collisions
- ▶ Rehash each bin with a new hash function for each bin
- ▶ This “second-layer” bin should have 0 collisions with high probability. . . how?
- ▶ If we hash  $n$  items to  $n^2$  buckets,  
$$\mathbb{E}[C] \leq \binom{n}{2} \frac{1}{n^2} \leq \frac{1}{2} \implies \mathbb{P}[C \geq 0] \leq \frac{1}{2}$$
- ▶ If the  $i^{\text{th}}$  entry of  $A$  has  $b_i$  items, then the second-layer hash table of the  $i^{\text{th}}$  entry has size  $b_i^2$

This is the FKS<sup>1</sup> scheme for perfect hashing for the static dictionary problem.

---

<sup>1</sup>Fredman, Kolmós, Szemerédi

## Analysis of FKS Hashing

- ▶ Total size of data structure is  $O(k)$  (for the first hash table) plus  $\sum_{i=1}^k b_i^2$  (for the second-layer hash tables) plus the cost to store the hash functions
- ▶ As we want to save space, we'd like  $\sum_{i=1}^k b_i^2 \in O(k)$
- ▶  $\sum_{i=1}^k b_i^2 = 2 \cdot \mathcal{C} + \sum_{i=1}^k b_i$  because  
$$\mathcal{C} = \sum_{i=1}^k \binom{b_i}{2} = \frac{1}{2} \sum_{i=1}^k b_i^2 - \frac{1}{2} \sum_{i=1}^k b_i$$
- ▶  $\mathbb{E}[\sum_{i=1}^k b_i^2] \leq 2 \mathbb{E}[\mathcal{C}] + k = 2 \binom{k}{2} \frac{1}{k} + k \leq 2k$
- ▶ Overall space is  $O(k)$ . To search, compute  $i = h(x)$  and find key in  $A_i[h_i(x)]$

## Summary

- ▶ Described a single hash function mapping from universe to bins and saw how it was implemented in CS 61B
- ▶ Secured ourselves against adversaries by choosing hash functions randomly from a family
- ▶ Drew analogy from balls and bins to “fully independent hashing” to understand collisions
- ▶ Compared the load balancing problem to hashing and found a bound for the length of the longest list and therefore an  $O(\cdot)$  expression for the expected worst-case performance.
- ▶ To conserve space while maintaining collision resistance, we designed a universal hash family
- ▶ Armed with all this we made the FKS “perfect hashing” scheme for static dictionaries where even the worst-case lookup is constant!