Lecture 4: Hashing

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

September 21, 2020

Overview

- Introduction
  - Hash Functions
  - Why is hashing?
  - How to hash?
- Succinctness of Hash Functions
  - Coping with randomness
  - Universal Hashing
  - Hashing using 2-universal families
  - Perfect Hashing
- Acknowledgements

Computational Model

Before we talk about hash functions, we need to state our model of computation:

Definition (Word RAM model)

In the word RAM[a] model:

- all elements are integers that fit in a machine word of w bits
- Basic operations (comparison, arithmetic, bitwise) on such words take $\Theta(1)$ time
- We can also access *any* position in the array in $\Theta(1)$ time

[a] RAM stands for Random Access Model

What is hashing?

We want to store n elements (keys) from the set $U = \{0, 1, \ldots, m-1\}$, where $m >> n$, in a data structure that supports *insertions, deletions, search* "as efficiently as possible."

$m >> n$ and $\log m < w$

(each element from $\{0, 1, \ldots, m-1\}$ fits in one word of memory)

What is hashing?

We want to store n elements (keys) from the set $U = \{0, 1, \ldots, m-1\}$, where $m >> n$, in a data structure that supports *insertions, deletions, search* "as efficiently as possible."

Naive approach: use an array A of m elements, initially $A[i] = 0$ for all i, and when a key is inserted, set $A[i] = 1$.

What is hashing?

We want to store n elements (keys) from the set $U = \{0, 1, \ldots, m-1\}$, where $m >> n$, in a data structure that supports *insertions, deletions, search* "as efficiently as possible."

Naive approach: use an array A of m elements, initially $A[i] = 0$ for all i, and when a key is inserted, set $A[i] = 1$.

- Insertion: $O(1)$, Deletion: $O(1)$, Search: $O(1)$
- Memory: $O(m \log(m))$ *(this is very bad!)*

>> our entire memory!

What is hashing?

We want to store n elements (keys) from the set $U = \{0, 1, \ldots, m-1\}$, where $m >> n$, in a data structure that supports *insertions, deletions, search* "as efficiently as possible."

Naive approach: use an array A of m elements, initially $A[i] = 0$ for all i, and when a key is inserted, set $A[i] = 1$.

- Insertion: $O(1)$, Deletion: $O(1)$, Search: $O(1)$
- Memory: $O(m \log(m))$ *(this is very bad!)*

Want to also achieve optimal memory $\boxed{O(n \log(m))}$. For this we will use a technique called *hashing*.

- A *hash function* is a function $h : U \to [0, n-1]$, where $|U| = m >> n$.
- A *hash table* is a data structure that consists of:
  - a table T with n cells $[0, n-1]$, each cell storing $O(\log(m))$ bits
  - a hash function $h : U \to [0, n-1]$

From now on, we will define memory as *# of cells*. "$w = O(\log m)$"

Why is hashing useful?

- Designing efficient data structures (dictionaries) for searching
- Data streaming algorithms
- Derandomization
- Cryptography
- Complexity Theory
- many more

Challenges in Hashing

Setup:

- Universe $U = \{0, \ldots, m-1\}$ of size $m >> n$ where n is the size of the range of our hash function $h : U \rightarrow [0, n-1]$
- Store $O(n)$ elements of U (keys) in hash table T (which has n cells)

Challenges in Hashing

Setup:

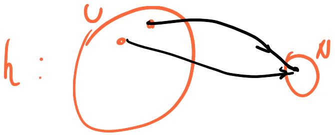
- Universe $U = \{0, \ldots, m-1\}$ of size $m >> n$ where n is the size of the range of our hash function $h : U \to [0, n-1]$
- Store $O(n)$ elements of U (keys) in hash table T (which has n cells)

Ideally, want hash function to map *different keys* into *different locations*.

Definition (Collision)

We say that a *collision* happens for hash function h with inputs $x, y \in U$ if $x \neq y$ and $h(x) = h(y)$.

By pigeonhole principle, impossible to achieve without knowing keys in advance.



Challenges in Hashing

Setup:

- Universe $U = \{0, \ldots, m-1\}$ of size $m >> n$ where n is the size of the range of our hash function $h : U \to [0, n-1]$
- Store $O(n)$ elements of U (keys) in hash table T (which has n cells)

Ideally, want hash function to map *different keys* into *different locations*.

Definition (Collision)

We say that a *collision* happens for hash function h with inputs $x, y \in U$ if $x \neq y$ and $h(x) = h(y)$.

By pigeonhole principle, impossible to achieve without knowing keys in advance.

Will settle for: # collisions *small with high probability*.

Our solution: family of hash functions

Construct *family* of hash functions $\mathcal{H}$ such that the *number of collisions* is **small** with **high probability**, when we pick hash function uniformly at random from the family $\mathcal{H}$.

Our solution: family of hash functions

Construct *family* of hash functions $\mathcal{H}$ such that the *number of collisions* is **small** with **high probability**, when we pick hash function uniformly at random from the family $\mathcal{H}$.

Simplest version to keep in mind:

$$\Pr_{h \in_R \mathcal{H}} [h(x) = h(y)] \leq \frac{1}{\text{poly}(n)} \quad \boxed{\forall x \neq y \in U}$$

collision

small

when take random function h from $\mathcal{H}$

For any pair $x \neq y \in U$ $\Pr_{\text{random } h}$ (x and y collide) is small

Our solution: family of hash functions

Construct *family* of hash functions $\mathcal{H}$ such that the *number of collisions* is **small** with **high probability**, when we pick hash function uniformly at random from the family $\mathcal{H}$.

Simplest version to keep in mind:

$$\Pr_{h \in_R \mathcal{H}}[h(x) = h(y)] \le \frac{1}{\mathrm{poly}(n)} \qquad \forall x \ne y \in U$$

Assumptions:

- keys are independent from hash function we choose.
- we **do not** know keys in advance (even if we did, nontrivial problem!)

Question

Still could have collisions. How do we handle them?

Random Hash Functions?

Natural to consider following approach:

From all functions $h : U \to \underbrace{[0, n-1]}_{N}$, just pick one uniformly at random.

Random Hash Functions?

Natural to consider following approach:

balls bins

From all functions $h : U \to [0, n-1]$, just pick one uniformly at random.

{0, 1, ..., m-1}

This setting is same as our balls-and-bins setting!

Random Hash Functions?

Natural to consider following approach:

From all functions $h : U \to [0, n-1]$, just pick one uniformly at random.

This setting is same as our balls-and-bins setting!
So, if we have to store n keys:

- Expected number of keys in a location: 1 (expected # balls in a bin)
- maximum number of collisions (max load) in one particular location: $O(\log n / \log \log n)$ keys

Random Hash Functions?

Natural to consider following approach:

From all functions $h : U \to [0, n-1]$, just pick one uniformly at random.

This setting is same as our balls-and-bins setting!
So, if we have to store n keys:

- Expected number of keys in a location: 1
- maximum number of collisions (max load) in one particular location: $O(\log n / \log \log n)$ keys

Solving collisions: store all keys hashed into location i by a linked list.

Known as *chain hashing*.

Random Hash Functions?

Natural to consider following approach:

From all functions $h : U \to [0, n-1]$, just pick one uniformly at random.

This setting is same as our balls-and-bins setting!
So, if we have to store n keys:

- Expected number of keys in a location: 1
- maximum number of collisions (max load) in one particular location: $O(\log n / \log\log n)$ keys

Solving collisions: store all keys hashed into location i by a linked list.

Known as *chain hashing*.

Could also pick *two* random hash functions and use *power of two choices*.
Collision bound becomes $\boxed{O(\log\log n)}$. $h_1(x)$ $h_2(x)$

Random Hash Functions?

Random hash functions look very good. However, we haven't discussed the following:

Random Hash Functions?

Random hash functions look very good. However, we haven't discussed the following:

Question

How much resource (time & space) does it take to compute random hash functions?

Random Hash Functions?

Random hash functions look very good. However, we haven't discussed the following:

Question

How much resource (time & space) does it take to compute random hash functions?

- Storing entire function $h : U \to [0, n-1]$ require $\tilde{O}(m \log n)$ bits (way too much space!) for each $x \in U$ $(x, h(x)) \to \log n$
- Even if we only stored the elements we saw, would require $\boxed{O(n)}$ time to evaluate $h(x)$ (need to decide if we had already computed it!)

$x_0, x_1, \ldots, x_{n-1}, x_n$

$x_{n-1} = x_0$

$(x_0, h(x_0))$

$(x_1, h(x_1))$

$h(x_{n-1}) = h(x_0)$

$\Theta(m)$ cells

Random Hash Functions?

Random hash functions look very good. However, we haven't discussed the following:

Question

How much resource (time & space) does it take to compute random hash functions?

- Storing entire function $h : U \to [0, n-1]$ require $O(m \log n)$ bits (way too much space!)
- Even if we only stored the elements we saw, would require $O(n)$ time to evaluate $h(x)$ (need to decide if we had already computed it!)

Remark

Thus, for random function all operations (insert, delete, search) take $O(n \log m)$ time (at best!)

in RAM $O(n)$ time!

Random Hash Functions?

Random hash functions look very good. However, we haven't discussed the following:

Question

How much resource (time & space) does it take to compute random hash functions?

- Storing entire function $h : U \to [0, n-1]$ require $O(m \log n)$ bits (way too much space!)
- Even if we only stored the elements we saw, would require $O(n)$ time to evaluate $h(x)$ (need to decide if we had already computed it!)

Remark

Thus, for random function all operations (insert, delete, search) take $O(n \log m)$ time (at best!)

How do we cope with the computational problem that arose with randomness?

How to cope with "hardness" of randomness?

We want something that is *random-like* (few collisions w.h.p.) but *easy to compute/represent*.

How to cope with "hardness" of randomness?

We want something that is *random-like* (few collisions w.h.p.) but *easy to compute/represent*.

Ideally something that takes $O(\log m)$ time to compute (as this is the size of our input).

linear time

How to cope with "hardness" of randomness?

We want something that is *random-like* (few collisions w.h.p.) but *easy to compute/represent*.

Ideally something that takes $O(\log m)$ time to compute (as this is the size of our input).

Question

How many hash functions can we have with the property above?

How to cope with "hardness" of randomness?

We want something that is *random-like* (few collisions w.h.p.) but *easy to compute/represent*.

Ideally something that takes $\boxed{O(\log m) \text{ time to}}$ compute (as this is the size of our input).

Question

How many hash functions can we have with the property above?

poly(m) functions, as each function takes at most $\boxed{O(\log m) \text{ bits}}$ to describe. Thus these are *succinct functions* (easy to describe and compute) which have *random-like* properties!

$2^{O(\log m)}$

How to cope with "hardness" of randomness?

We want something that is *random-like* (few collisions w.h.p.) but *easy to compute/represent*.

Ideally something that takes $O(\log m)$ time to compute (as this is the size of our input).

Question

How many hash functions can we have with the property above?

poly(m) functions, as each function takes at most $O(\log m)$ bits to describe. Thus these are *succinct functions* (easy to describe and compute) which have *random-like* properties!

Part of *derandomization/pseudorandomness*: huge subfield in TCS!

k-wise independence

Weaker notion of independence.

k-wise independence

Weaker notion of independence.

Definition (Full Independence)

A set of random variables $X_1, \ldots, X_n$ are said to be (fully) independent if they satisfy

$$\Pr\left[\bigcap_{i=1}^{n} X_i = a_i\right] = \prod_{i=1}^{n} \Pr[X_i = a_i]$$

$X_1 = a_1, X_2 = a_2 \ldots X_n = a_n$

k-wise independence

Weaker notion of independence.

Definition (Full Independence)

A set of random variables $X_1, \ldots, X_n$ are said to be (fully) independent if they satisfy

$$\Pr\left[\bigcap_{i=1}^{n} X_i = a_i\right] = \prod_{i=1}^{n} \Pr[X_i = a_i] \quad \times$$

Definition (k-wise Independence)

A set of random variables $X_1, \ldots, X_n$ are said to be k-wise independent if for any set $\boxed{J \subset [n]}$ such that $\underline{|J| \le k}$ they satisfy

only small sets variables behave as if independent

$$\Pr\left[\bigcap_{i \in J} X_i = a_i\right] = \prod_{i \in J} \Pr[X_i = a_i]$$

Pairwise independence

When $k = 2$, k-wise independence is called *pairwise independence*.

Example (XOR pairwise independence)

Given b uniformly random bits $Y_1, \ldots, Y_b$, we can generate $2^b - 1$ pairwise independent random variables as follows:

$$X_S := \bigoplus_{i \in S} Y_i \qquad \boxed{S \subseteq [b] \cancel{\setminus \emptyset} \quad S \neq \emptyset}$$

$a \oplus b$

a	b	$a \oplus b$
0	0	0
1	0	1
0	1	1
1	1	0

Pairwise independence

When $k = 2$, k-wise independence is called *pairwise independence*.

Example (XOR pairwise independence)

Given b uniformly random bits $Y_1, \ldots, Y_b$, we can generate $2^b - 1$ pairwise independent random variables as follows:

$$X_S := \bigoplus_{i \in S} Y_i \qquad S \subseteq [b] \setminus \emptyset$$

Pairwise independence

When $k = 2$, k-wise independence is called *pairwise independence*.

Example (XOR pairwise independence)

Given b uniformly random bits $Y_1, \ldots, Y_b$, we can generate $2^b - 1$ pairwise independent random variables as follows:

$$X_S := \bigoplus_{i \in S} Y_i \qquad S \subseteq [b] \setminus \emptyset$$

- Why are they even random?

$$X_{\{1,2,3\}} = \underbrace{Y_1 \oplus Y_2}_{1} \oplus \underbrace{Y_3}_{\substack{1 \to 0 \\ 0 \to 1}}$$

$$X_S = \begin{cases} 1 & 1/2 \\ 0 & 1/2 \end{cases}$$

Pairwise independence

When $k = 2$, k-wise independence is called *pairwise independence*.

Example (XOR pairwise independence)

Given b uniformly random bits $Y_1, \ldots, Y_b$, we can generate $2^b - 1$ pairwise independent random variables as follows:

$$X_S := \bigoplus_{i \in S} Y_i \qquad S \subseteq [b] \setminus \emptyset$$

- Why are they even random?
- Why are they pairwise independent?

$S_1 \neq S_2$ there is variable

$Y_1 \in S_1 \setminus S_2$

S_1, S_2 subsets of $[b]$

$$\Pr[X_{S_1} = a_1 \text{ and } X_{S_2} = a_2] = \frac{1}{4}$$

Pairwise independence

When $k = 2$, k-wise independence is called *pairwise independence*.

Example (XOR pairwise independence)

Given b uniformly random bits $Y_1, \ldots, Y_b$, we can generate $2^b - 1$ pairwise independent random variables as follows:

$$X_S := \bigoplus_{i \in S} Y_i \qquad S \subseteq [b] \setminus \emptyset$$

- Why are they even random?
- Why are they pairwise independent?
- Are they also 3-wise independent?

No $X_{\{1\}}$ $X_{\{2\}}$ $X_{\{1,2\}}$ not 3-wise ind.

Pairwise independence II

integers mod p.

Example (Pairwise independence in $\mathbb{F}_p$)

Let p be a prime number. Given 2 uniformly random variables $Y_1, Y_2 \sim \underbrace{[0, \ldots, p-1]}_{\mathbb{F}_p}$, generate p pairwise independent random variables as follows:

$$X_i := Y_1 + i \cdot Y_2 \mod p \qquad i \in [0, p-1]$$

Pairwise independence II

Example (Pairwise independence in $\mathbb{F}_p$)

Let p be a prime number. Given 2 uniformly random variables $Y_1, Y_2 \sim [0, \ldots, p-1]$, generate p pairwise independent random variables as follows:

$$X_i := Y_1 + i \cdot Y_2 \mod p \qquad i \in [0, p-1]$$

- Why are they even random?

$$X_0 = Y_1$$

$$X_1 = Y_1 + Y_2 \quad \rightsquigarrow \quad \Pr[X_i = a] = \frac{1}{p}$$

Pairwise independence II

Example (Pairwise independence in $\mathbb{F}_p$)

Let p be a prime number. Given 2 uniformly random variables $Y_1, Y_2 \sim [0, \ldots, p-1]$, generate p pairwise independent random variables as follows:

$$X_i := Y_1 + i \cdot Y_2 \mod p \qquad i \in [0, p-1]$$

- Why are they even random?
- Why are they pairwise independent?

X_i , X_j
$\shortparallel \quad \shortparallel$
$a \quad b$

$Y_1 + i Y_2 = a$
$Y_1 + j Y_2 = b$

1 solution

$$\boxed{\Pr[X_i = a, X_j = b] = \frac{1}{p^2}} \qquad \det\begin{pmatrix} 1 & i \\ 1 & j \end{pmatrix} = j - i \text{ (invertible)}$$

$\shortparallel$
$\Pr[X_i = a] \cdot \Pr[X_j = b]$

Pairwise independence II

Example (Pairwise independence in $\mathbb{F}_p$)

Let p be a prime number. Given 2 uniformly random variables $Y_1, Y_2 \sim [0, \ldots, p-1]$, generate p pairwise independent random variables as follows:

$$X_i := Y_1 + i \cdot Y_2 \mod p \qquad i \in [0, p-1]$$

- Why are they even random?
- Why are they pairwise independent?
- Are they also 3-wise independent?

NO. Practice problem.

Pairwise independence II

Example (Pairwise independence in $\mathbb{F}_p$)

Let p be a prime number. Given 2 uniformly random variables $Y_1, Y_2 \sim [0, \ldots, p-1]$, generate p pairwise independent random variables as follows:

$$X_i := Y_1 + i \cdot Y_2 \mod p \qquad i \in [0, p-1]$$

- Why are they even random?
- Why are they pairwise independent?
- Are they also 3-wise independent?

Can think of these random variables as picking a random line over a finite field. If we only know one point of the line, the second point is still uniformly random. However two points determine the line.

Universal Hash Functions

We want hash functions. Why are we talking about random variables?

Universal Hash Functions

We want hash functions. Why are we talking about random variables?

Definition (Universal Hash Functions)

Let U be a universe with $|U| \geq n$. A family of hash functions $\mathcal{H} = \{h : U \to [0, n-1]\}$ is *k-universal* if, *for any* **distinct** elements $u_1, \ldots, u_k \in U$, we have

$$\Pr_{h \in_R \mathcal{H}} [h(u_1) = h(u_2) = \ldots = h(u_k)] \leq 1/n^{k-1}$$

Universal Hash Functions

We want hash functions. Why are we talking about random variables?

Definition (Universal Hash Functions)

Let U be a universe with $|U| \geq n$. A family of hash functions $\mathcal{H} = \{h : U \to [0, n-1]\}$ is *k-universal* if, *for any* **distinct** elements $u_1, \ldots, u_k \in U$, we have

$$\Pr_{h \in_R \mathcal{H}} [h(u_1) = h(u_2) = \ldots = h(u_k)] \leq 1/n^{k-1}$$

Definition (Strongly Universal Hash Functions)

$\mathcal{H} = \{h : U \to [0, n-1]\}$ is *strongly k-universal* if, *for any* **distinct** elements $u_1, \ldots, u_k \in U$ and *for any* values $y_1, \ldots, y_k \in [0, n-1]$, we have

$$\Pr_{h \in_R \mathcal{H}} [\underline{h(u_1) = y_1}, \ldots, \underline{h(u_k) = y_k}] \leq 1/n^k$$

random like.

Relation to k-wise independent random variables

What do the previous definitions have to do with random variables?

Relation to k-wise independent random variables

What do the previous definitions have to do with random variables?

Family $\mathcal{H}$ is *strongly k-universal* if the random variables $h(0), \ldots, h(|U| - 1)$ are *k-wise independent*.

Relation to k-wise independent random variables

What do the previous definitions have to do with random variables?

Family $\mathcal{H}$ is *strongly k-universal* if the random variables $h(0), \ldots, h(|U| - 1)$ are *k-wise independent*.

Can use random variables to construct universal hash functions!

Strongly 2-universal families of hash functions

Let p be a prime number, $U = [0, p-1]$.

Proposition

$$\mathcal{H} = \{h_{a,b}(\underbrace{x}_{\in U}) := a \cdot x + b \mod p \mid a, b \in [0, p-1]\}$$

is strongly 2-universal.

Proof xc $a = Y_1$ $b = Y_2$ random vars proof

Strongly 2-universal families of hash functions

Let p be a prime number, $U = [0, p-1]$.

Proposition

$$\mathcal{H} = \{h_{a,b}(x) := a \cdot x + b \mod p \quad | \quad a, b \in [0, p-1]\}$$

is strongly 2-universal.

How do we make the domain U much larger than image of the maps? (as usually in hashing size of universe much larger than size of table)

Strongly 2-universal families of hash functions

Let p be a prime number, $U = [0, p-1]. = \mathbb{F}_p$

Proposition

$$\mathcal{H} = \{h_{a,b}(x) := a \cdot x + b \mod p \quad | \quad a, b \in [0, p-1]\}$$

is strongly 2-universal.

How do we make the domain U much larger than image of the maps? (as usually in hashing size of universe much larger than size of table)

Proposition

Let $U = [0, p^k - 1] \equiv \underbrace{[0, p-1]^k \setminus \{(0, \ldots, 0)\}}_{\mathbb{F}_p^k \setminus \vec{0}}$ *and* $\mathbf{a} = (a_0, \ldots, a_{k-1}) \in U$

$$\mathcal{H} = \{h_{\mathbf{a},b}(\mathbf{x}) := \underbrace{\mathbf{a} \cdot \mathbf{x}}_{a_0 \cdot x_0 + \cdots + a_k \cdot x_k} + b \mod p \quad | \quad \mathbf{a} \in U, b \in [0, p-1]\}$$

$h : \mathbb{F}_p^k \to \mathbb{F}_p$

is strongly 2-universal.

Strongly 2-universal families of hash functions

Proposition

Let $U = [0, p^k - 1] \equiv [0, p-1]^k \setminus \{(0, \ldots, 0)\}$ *and* $\mathbf{a} = (a_0, \ldots a_{k-1})$

$$\mathcal{H} = \{h_{\mathbf{a},b}(\mathbf{x}) := \mathbf{a} \cdot \mathbf{x} + b \mod p \quad | \quad \mathbf{a} \in U, b \in [0, p-1]\}$$

is strongly 2-universal.

2-universal families of hash functions

What if my hast table size is not a prime?

Proposition

$$\mathcal{H} = \{h_{a,b}(x) := (a \cdot x + b \mod p) \mod n \quad | \quad a, b \in [0, p-1]\}$$

is 2-universal (but not strongly 2-universal).

Practice problem: prove the proposition above.

k-universal families of hash functions

Can we construct k-universal families of hash functions like this?

k-universal families of hash functions

Can we construct k-universal families of hash functions like this?

- YES! Instead of constructing random lines (degree 1 polynomials), can construct random univariate polynomials of degree $k-1$

$ax+b$

$$\boxed{a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1 x + a_0}$$

k-universal families of hash functions

Can we construct k-universal families of hash functions like this?

- YES! Instead of constructing random lines (degree 1 polynomials), can construct random univariate polynomials of degree $k - 1$
- Two points determine a line. Similarly, k points determine a univariate polynomial of degree $k - 1$

k-universal families of hash functions

Can we construct k-universal families of hash functions like this?

- YES! Instead of constructing random lines (degree 1 polynomials), can construct random univariate polynomials of degree $k-1$
- Two points determine a line. Similarly, k points determine a univariate polynomial of degree $k-1$
- Random degree $k-1$ polynomials are k-wise independent!
- Practice problem: prove this!

Efficiency

How did pairwise independent improve the problems we were having with random functions?

Efficiency

How did pairwise independent improve the problems we were having with random functions?

Remark

For random function all operations (insert, delete, search) take $O(n \log m)$ time (at best!)

Efficiency

How did pairwise independent improve the problems we were having with random functions?

Remark

For random function all operations (insert, delete, search) take $O(n \log m)$ time (at best!)

Remark

- In XOR example, our function takes $O(n)$ storage space, and $O(n)$ time to compute.[a]
- In $\mathbb{F}_p$ examples, our function takes $O(1)$ storage space and $O(1)$ time to compute![b]

$h_{a,b}$ store a, b $O(1)$

$ax+b$ $O(1)$ time

[a] Reminder that we assume that $n < 2^w$.
[b] We assume that $p < 2^w$.

Theorem from Probability

Theorem (Markov's inequality)

If X is a non-negative random variable and $t > 0$, we have:

$$\Pr[X \geq t] \leq \frac{\mathbb{E}[X]}{t}$$

Hashing with 2-universal families

- Let $U = [0, m-1]$, and p be a prime number such that $m \leq p < 2m$ (exists by Bertrand's postulate)

Hashing with 2-universal families

- Let $U = [0, m-1]$, and p be a prime number such that $m \leq p < 2m$ (exists by Bertrand's postulate)
- $\mathcal{H} = \{h_{a,b}(x) := (a \cdot x + b \mod p) \mod n \quad | \quad a, b \in [0, p-1]\}$

Hashing with 2-universal families

- Let $U = [0, m-1]$, and p be a prime number such that $m \leq p < 2m$ (exists by Bertrand's postulate)
- $\mathcal{H} = \{h_{a,b}(x) := (a \cdot x + b \mod p) \mod n \quad | \quad a, b \in [0, p-1]\}$
- Only need to choose $a, b \in [0, p-1]$ to store a function from $\mathcal{H}$.

Hashing with 2-universal families

- Let $U = [0, m-1]$, and p be a prime number such that $m \leq p < 2m$ (exists by Bertrand's postulate)
- $\mathcal{H} = \{h_{a,b}(x) := (a \cdot x + b \mod p) \mod n \quad | \quad a, b \in [0, p-1]\}$
- Only need to choose $a, b \in [0, p-1]$ to store a function from $\mathcal{H}$.
- Computation time of $h_{a,b}$ is also $O(\log m)$

Hashing with 2-universal families

- Let $U = [0, m-1]$, and p be a prime number such that $m \leq p < 2m$ (exists by Bertrand's postulate)
- $\mathcal{H} = \{h_{a,b}(x) := (a \cdot x + b \mod p) \mod n \quad | \quad a, b \in [0, p-1]\}$
- Only need to choose $a, b \in [0, p-1]$ to store a function from $\mathcal{H}$.
- Computation time of $h_{a,b}$ is also $O(\log m)$
- Can this hash function match chain hashing parameters?
  ($O(\log \log n)$ search time)

Hashing with 2-universal families

- Let $U = [0, m-1]$, and p be a prime number such that $m \leq p < 2m$ (exists by Bertrand's postulate)
- $\mathcal{H} = \{h_{a,b}(x) := (a \cdot x + b \mod p) \mod n \quad | \quad a, b \in [0, p-1]\}$
- Only need to choose $a, b \in [0, p-1]$ to store a function from $\mathcal{H}$.
- Computation time of $h_{a,b}$ is also $O(\log m)$
- Can this hash function match chain hashing parameters?
  ($O(\log \log n)$ search time)

Do not have same expected search time as chain hashing.

Hashing with 2-universal families

- Let $U = [0, m-1]$, and p be a prime number such that $m \leq p < 2m$ (exists by Bertrand's postulate)
- $\mathcal{H} = \{h_{a,b}(x) := (a \cdot x + b \mod p) \mod n \quad | \quad a, b \in [0, p-1]\}$
- Only need to choose $a, b \in [0, p-1]$ to store a function from $\mathcal{H}$.
- Computation time of $h_{a,b}$ is also $O(\log m)$
- Can this hash function match chain hashing parameters? ($O(\log\log n)$ search time)

Do not have same expected search time as chain hashing.

Lemma (Maximum number of collisions)

The expected number of collisions using a 2-universal hash family is

$$\ell^2/2n$$

ℓ = # elements that will hash

Hashing with 2-universal families

ℓ = # elements from U that we will hash

Lemma (Maximum number of collisions)

The expected number of collisions using a 2-universal hash family is

$$\leq \ell^2/2n$$

$$X_{ij} = \begin{cases} 1 & \text{if keys } i \text{ and } j \text{ are mapped to same location} \\ 0 & \text{otherwise} \end{cases}$$

$$X = \sum_{1 \leq i < j \leq \ell} X_{ij} \quad \text{\# collision pairs}$$

$$E[X] = \sum_{i,j} E[X_{ij}] = \sum_{i,j} \frac{1}{n} \leq \frac{\ell^2}{2n}$$

2-universal

By Markov we have that

$$\Pr\left[X > \underbrace{\ell^2/n}_{\text{many collisions}}\right] < \frac{1}{2}$$

Hashing with 2-universal families

Lemma (Maximum number of collisions)

The expected number of collisions using a 2-universal hash family is

$$\ell^2/2n$$

Thus, by Markov's inequality, we have $\frac{t^2}{2} \sim \binom{t}{2} \leq \frac{\ell^2}{n}$ with prob. $> 1/2$

Lemma (Maximum load of entry of hash table)

With probability $\geq 1/2$ the ~~number of collisions~~ max load of cell using a 2-universal hash family is

$$\leq \sqrt{\frac{2\ell^2}{n}}.$$

$t \leq \sqrt{\frac{2\ell^2}{n}}$

When $\ell \approx n$ (as is usually assumed in hashing), we expect $\sqrt{2n}$. collisions

t entries in a cell $\Rightarrow$ have $\geq \binom{t}{2}$ collisions

Perfect Hashing

How to build a hash table with $O(1)$ search time and $O(n)$ memory? Can we still do it with a 2-universal family of hash functions?

Setup: we know the keys in advance (static data structure)

(in lecture: how can we use 2-universal hashing to get $O(1)$ search time but that required $O(n^2)$ memory)

Perfect Hashing

How to build a hash table with $O(1)$ search time and $O(n)$ memory? Can we still do it with a 2-universal family of hash functions?

Corollary

If $h \in \mathcal{H}$ is a random hash function from a 2-universal family of hash functions, then for any set $S \subseteq U$ of size $\ell \leq \sqrt{n}$, the probability of h being perfect for S is at least $1/2$.

no collisions on S

Proof: There is no collision with probability $\geq 1/2$.

Perfect Hashing

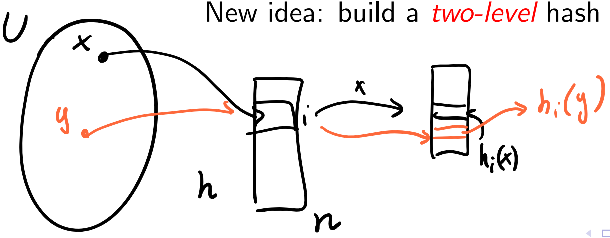
How to build a hash table with $O(1)$ search time and $O(n)$ memory? Can we still do it with a 2-universal family of hash functions?

Corollary

If $h \in \mathcal{H}$ is a random hash function from a 2-universal family of hash functions, then for any set $S \subseteq U$ of size $\ell \leq \sqrt{n}$, the probability of h being perfect for S is at least $1/2$.

Proof: There is no collision with probability $\geq 1/2$.

New idea: build a *two-level* hash table!



Perfect Hashing

How to build a hash table with $O(1)$ search time and $O(n)$ memory? Can we still do it with a 2-universal family of hash functions?

Corollary

If $h \in \mathcal{H}$ is a random hash function from a 2-universal family of hash functions, then for any set $S \subseteq U$ of size $\ell \leq \sqrt{n}$, the probability of h being perfect for S is at least $1/2$.

Proof: There is no collision with probability $\geq 1/2$.

New idea: build a *two-level* hash table!

Theorem

The two-level approach gives perfect hashing scheme.

how many hash functions have we used? ($O(n)$) ✓

memory is also $O(n)$

Proof (sketch) of Theorem

Remark we know keys in advance S set of keys

Approach: pick first layer hash function h uniformly at random. Test h on S.

With probability $\geq 1/2$ max # collisions in one bin is $\leq \sqrt{n}$. We will get good hash fcn h with constant # tries.

Assume that max # collisions (h, S) is $\leq \sqrt{n}$

$\ell_i \leftarrow$ load at i^{th} cell of hash table given by h.

know : $l_i \le \sqrt{n}$ (because h is good for S)

and $\sum_{i=0}^{n-1} l_i = n$ $(= |S|)$

Lemma if take h_i random hash function from $\boxed{h_i : S \to l_i^2}$ h_i is perfect for the l_i elements that map into i^{th} cell of first table h.

We showed that expected # collisions is 0.

Bound on Memory: $\sum_{i=0}^{n-1} l_i^2$ (bad because $\sqrt{n} \cdot (\sqrt{n})^2 = n^{3/2}$)

We also know (because h is good for S)

w.h.p. # collisions is $\leq \ell^2/n$

(expected # collisions is $\leq \ell^2/2n$ ← lemma)

when $\ell = n$ (our case) we get that

collisions $\leq n$

$$\sum_{i=0}^{n-1} \ell_i^2 = O(\text{\# collisions of } h)$$

Acknowledgement

- Lecture based largely on Lap Chi's notes.
- See Lap Chi's notes at `https://cs.uwaterloo.ca/~lapchi/cs466/notes/L05.pdf`

References I

Motwani, Rajeev and Raghavan, Prabhakar (2007)
Randomized Algorithms

Mitzenmacher, Michael, and Eli Upfal (2017)
Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis.
Cambridge university press, 2017.