Lecture 5: Hashing

Rafael Oliveira

University of Waterloo
Cheriton School of Computer Science

rafael.oliveira.teaching@gmail.com

May 25, 2023

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 ℓ elements (keys) from the set $U = \{0, 1, \ldots, m-1\}$, where $2^w > m \gg \ell$, in a data structure that supports *insertions, deletions, search* "as efficiently as possible."

What is hashing?

We want to store ℓ elements (keys) from the set $U = \{0, 1, \ldots, m-1\}$, where $2^w > m \gg \ell$, 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 ℓ elements (keys) from the set $U = \{0, 1, \ldots, m-1\}$, where $2^w > m \gg \ell$, 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: $\Theta(m)$ *(this is very bad!)*

What is hashing?

We want to store ℓ elements (keys) from the set $U = \{0, 1, \ldots, m-1\}$, where $2^w > m \gg \ell$, 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: $\Theta(m)$ *(this is very bad!)*

Want to also achieve optimal memory $O(\ell)$. For this we will use a technique called *hashing*.

- A *hash function* is a function $h : U \rightarrow [0, n-1]$, where $|U| = m \gg n$.
- A *hash table* is a data structure that consists of:
  - a table T with n cells $[0, n-1]$,
  - a hash function $h : U \rightarrow [0, n-1]$

From now on, we will define memory as *# of cells*.

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

Challenges in Hashing

Setup:

- Universe $U = \{0, \ldots, m-1\}$ of size $m \gg n$ where n is the size of the range of our hash function $h : U \to [0, n-1]$
- Store $\ell := O(n)$ 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 $m \gg n$ where n is the size of the range of our hash function $h : U \to [0, n-1]$
- Store $\ell := O(n)$ 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:

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

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:

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

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

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!

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 (i.e., if $\ell = n$):

- 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

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 (i.e., if $\ell = n$):

- 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 (i.e., if $\ell = n$):

- 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 $O(\log \log n)$.

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 $O(m \log n)$ bits (way too much space!)
- Even if we only stored the elements we saw, would require $O(\ell)$ time to evaluate $h(x)$ (need to decide if we had already computed it!)

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 $O(\ell)$ 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 $O(\ell)$ time (at best!). Since we are aiming for $\ell = O(n)$, then the time would be $O(n)$!

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 $O(\ell)$ 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 $O(\ell)$ time (at best!). Since we are aiming for $\ell = O(n)$, then the time would be $O(n)$!

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 $O(1)$ time to compute (as the size of our input is $O(\log m) = O(w) = O(1)$ in the RAM model).

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 $O(1)$ time to compute (as the size of our input is $O(\log m) = O(w) = O(1)$ in the RAM model).

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 $O(1)$ time to compute (as the size of our input is $O(\log m) = O(w) = O(1)$ in the RAM model).

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!

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 $O(1)$ time to compute (as the size of our input is $O(\log m) = O(w) = O(1)$ in the RAM model).

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

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

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

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

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 $J \subset [n]$ such that $|J| \leq k$ they satisfy

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

Pairwise independence

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

Example (XOR pairwise independence)

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

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

Pairwise independence

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

Example (XOR pairwise independence)

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

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

- Why are they even random?

Pairwise independence

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

Example (XOR pairwise independence)

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

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

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

Pairwise independence

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

Example (XOR pairwise independence)

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

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

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

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:

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

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?

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?

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?

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 $\mathcal{H} = \{h : U \rightarrow [0, n-1]\}$ 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)

$\mathcal{H} = \{h : U \rightarrow [0, n-1]\}$ 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

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

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

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

is strongly 2-universal.

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, $U = [0, p-1]$.

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

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

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 $k - 1$

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 independence improve the problems we were having with random functions?

Efficiency

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

Remark

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

Efficiency

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

Remark

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

Remark

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

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

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 $O(1)$

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 $\mathcal{H}$.
- Computation time of $h_{a,b}$ is also $O(1)$
- Can this hash function match chain hashing parameters? ($O(\log \log n)$ max load - which implies 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 $O(1)$
- Can this hash function match chain hashing parameters? ($O(\log \log n)$ max load - which implies 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 $O(1)$
- Can this hash function match chain hashing parameters? ($O(\log \log n)$ max load - which implies search time)

Do not have same expected search time as chain hashing.

Lemma (Maximum number of collisions)

Let our set of keys S be of size ℓ, and our hash functions be from U to $[0, n-1]$. The expected number of collisions, using a 2-universal hash family is

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

Hashing with 2-universal families

Lemma (Maximum number of collisions)

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

$$\ell^2/2n$$

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

Lemma (Maximum load of entry of hash table)

With probability $\geq 1/2$ *the number of collisions using a 2-universal hash family is*

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

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

Perfect Hashing

Setting: (*static keys*) Suppose now we are given the set S of keys in advance, and $|S| = n$ (so, $\ell = n$ here).
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?

Perfect Hashing

Setting: (*static keys*) Suppose now we are given the set S of keys in advance, and $|S| = n$ (so, $\ell = n$ here).
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 $\geq 1/2$.

Perfect Hashing

Setting: (*static keys*) Suppose now we are given the set S of keys in advance, and $|S| = n$ (so, $\ell = n$ here).
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

Setting: (*static keys*) Suppose now we are given the set S of keys in advance, and $|S| = n$ (so, $\ell = n$ here).
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.

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.