Universal Hashing and Perfect Hashing

Junhao Gan Tony Wirth

School of Computing and Information Systems
The University of Melbourne

March 28, 2022

Reading Materials

- Book Chapter by Jeff Erickson:
  http://jeffe.cs.illinois.edu/teaching/algorithms/notes/05-hashing.pdf
- Lecture Note by Erik Demaine:
  https://courses.csail.mit.edu/6.851/spring07/erik/L11.pdf
- Slides by Yufei Tao:
  http://www.cse.cuhk.edu.hk/~taoyf/course/comp3506/lec/hashing.pdf

Dictionary Search

Dictionary Search

In this lecture, we discuss the following problem:

Dictionary Search. Consider a set S of n elements from a universe $\mathcal{U}$; given a *lookup* query element $q \in \mathcal{U}$, decide whether or not q is in S.

In the following, we first consider the case that S is static, i.e., no insertions or deletions of the elements; and S remains the same for all *lookup* queries.

Without loss of generality, we assume that $\mathcal{U}$ is a set of integers, i.e., each element is an integer.

Think:

Any idea to solve this problem? What are the pre-processing time, space consumption and *lookup* query time complexities of your solution?

Static Dictionary Search

Possible Solutions

- **Solution 1.** Use an array of length $|\mathcal{U}|$: if the i^{th} element of $\mathcal{U}$ is in S, set $A[i] = 1$; otherwise, set $A[i] = 0$.
  - Pre-processing Time: $O(|\mathcal{U}|)$ worst-case
  - Space Consumption: $O(|\mathcal{U}|)$ worst-case
  - Query Time: $O(1)$ worst-case

Unfortunately, $|\mathcal{U}|$ can be very large (and sometimes even unbounded); it is often far larger than n.

Both the $O(|\mathcal{U}|)$ pre-processing time and space consumption are considered inefficient.

Static Dictionary Search

Possible Solutions

- **Solution 2.** Sort and store S in an array of length n; perform binary search on the sorted array to answer a query.
  - Pre-processing Time: $O(n \log n)$ worst-case
  - Space Consumption: $O(n)$ worst-case
  - Query Time: $O(\log n)$ worst-case

The above bounds can also be achieved by any deterministic balanced binary search trees.

Static Dictionary Search

A Desired Solution

Our goal is to achieve:

- Pre-processing Time: $O(n)$ expected
- Space Consumption: $O(n)$ worst-case
- Query Time: $O(1)$ worst-case

The technique deployed in the above solution is *Perfect Hashing* and the data structure used is the *Hash Table*.

Static Dictionary Search

A Gadget Solution

As the first step, we show a gadget solution that achieves:

- Pre-processing Time: $O(n + m)$ worst-case
- Space Consumption: $O(n + m)$ worst-case
- Query Time: $O(1 + \frac{n}{m})$ expected

where m is a *space budget* parameter that controls the trade-off between space consumption and query time.

The technique here is Universal Hashing.

Hash Functions and Hash Tables

Hashing

Hash Functions

Consider a space budget $m > 0$ which is typically far smaller than $|\mathcal{U}|$.

A *hash function* h is a function that maps the elements in $\mathcal{U}$ to an integer domain $\{0, 1, 2, \ldots, m-1\}$, formally written as:

$$h : \mathcal{U} \to \{0, 1, 2, \ldots, m-1\}\,.$$

For an element $x \in \mathcal{U}$, $h(x)$ is called the hash value of x (under h).

If two distinct elements x and y have the same hash value, i.e., $h(x) = h(y)$, we say that x and y *collide* (under h).

Hashing

Hash Tables

A *hash table* of S with respect to a hash function h is an array, denoted by T_h, of length m.

In particular, each cell $T_h[i]$, for $i = 0, 1, \ldots, m-1$, in T_h stores a certain data structure that maintains:

all the elements x in S having hash value i, i.e., $h(x) = i$.

In general, the data structure at each cell $T_h[i]$ is a *linked list*, denoted by L_i.

Each node in L_i stores an element x with $h(x) = i$ and each element x can be stored exactly once.

Hashing

Constructing a Hash Table

Given a set S of n elements and a hash function h, the hash table T_h can be constructed as follows:

- Create an array T_h of length m (i.e., with m cells).
- Initialize each cell $T_h[i]$ to have an empty linked list L_i.
- For each element $x \in S$, append x to the linked list $L_{h(x)}$, stored in $T_h[h(x)]$.

Analysis

Assuming that the hash value $h(x)$ for every element $x \in \mathcal{U}$ can be computed in $O(1)$ time, the worst-case construction time of T_h is $O(n + m)$.

The space consumption of T_h is $m + \sum_{i=0}^{m-1} |L_i| = O(n + m)$, where $|L_i|$ is the number of elements stored in L_i.

Hashing

Hand-Written Example 1

Consider $S = \{17, 19, 4, 77, 63, 86, 99\}$. Let $m = 5$ and $h(x) = x \bmod m$.

Hashing

Answering a Query

Given a query element q, the query algorithm is as follows:

- Compute $h(q)$;
- Scan the linked list $L_{h(q)}$ stored in $T_h[h(q)]$ and report whether q is found in $L_{h(q)}$.

The query time is bounded by $O(1 + |L_{h(q)}|)$, which depends on the length of $L_{h(q)}$.

Hashing

Answering a Query

Different hash functions would produce hash tables with different qualities.

In general, a good hash table should keep the each linked list short.

In this sense, a good hash function should be able to “spread” the elements out as even as possible.

Hashing

Hand-Written Example 2

Consider $S = \{17, 19, 4, 77, 63, 86, 99\}$. Let $m = 5$ and $h(x) = 4$.

Hashing

Unavoidable Worst Case

Unfortunately, by the *Pigeonhole Principle*:

For *each* fixed hash function h, there exists a set S of *at least* $\lfloor \frac{|\mathcal{U}|}{m} \rfloor$ "bad" elements such that all of them have the same hash value.

In other words, if the hash function h is known to an adversary, the adversary can always construct a sub-set S of all bad elements, on which h becomes completely useless!

Randomness is a key to remedy this issue, by which achieving good expected performance becomes possible.

Universal Hashing

Hash Function Family

More specifically, the rationale is as follows:

- the element set S is given *in advance* (this is the case in the static dictionary problem); or
- the hash function is kept *unknown* to adversaries (this is a common assumption in practice);
- if the hash function h is picked uniformly at random from a certain hash-function family $\mathcal{H}$, then the probability that h acting on S produces a badly distributed hash table is significantly reduced.

However, not every hash function family $\mathcal{H}$ allows us to achieve a good bound on the query time.

We need $\mathcal{H}$ to possess certain properties.

Universal Hashing

Universal Hashing

Universal Hash Function Family

A hash function family $\mathcal{H}$ is *universal* (more specifically, *two-universal*) if it has the following property.

If we uniformly at random pick a hash function h from $\mathcal{H}$, then:

for every pair of distinct elements x and y in $\mathcal{U}$, the probability that x and y collide under h satisfies:

$$\mathrm{Pr}_{h \in \mathcal{H}} [h(x) = h(y)] \leq \frac{1}{m}.$$

Universal Hashing

Universal Hash Function Family

We next show that the expected length of each linked list in the hash table T_h with respect to h drawn uniformly at random from a universal $\mathcal{H}$ is $O(1 + \frac{n}{m})$.

In particular, when $m = \Theta(n)$, the expected length is bounded by $O(1)$.

Universal Hashing

Bounding the Expected Length

Consider an arbitrary element $x \in \mathcal{U}$. For every element $y \in \mathcal{U}$, define an indicator variable $C_{x,y}$ such that:

$$C_{x,y} = 1 \textit{ if and only if } h(x) = h(y).$$

Consider the linked list $L_{h(x)}$ (into which x is hashed by h).Then we have:

$$|L_{h(x)}| = \sum_{y \in S} C_{x,y}.$$

Therefore, by the linearity of expectation:

$$E\left[|L_{h(x)}|\right] = \sum_{y \in S} E\left[C_{x,y}\right] = \sum_{y \in S} \Pr\left[h(x) = h(y)\right]. \tag{1}$$

Bounding the Expected Length

By the fact that h is uniformly at random chosen from a universal hash function family $\mathcal{H}$,

$$\Pr[h(x) = h(y)] = \begin{cases} 1 & \text{if } x = y \\ 1/m & \text{if } x \neq y \end{cases} \tag{2}$$

- Case 1: $x \in S$. Substituting (2) to (1), we have:

$$E\left[|L_{h(x)}|\right] = 1 + \sum_{y \neq x \in S} 1/m = 1 + \frac{n-1}{m}.$$

- Case 2: $x \notin S$ (this happens when x is a query element). We have:

$$E\left[|L_{h(x)}|\right] = \sum_{y \neq x \in S} 1/m = \frac{n}{m}.$$

Either way, $E\left[|L_{h(x)}|\right] \leq 1 + \frac{n}{m}$.

Universal Hashing

A Summary

Given a set S of n elements, with a hash function h uniformly at random chosen from a universal family $\mathcal{H}$, we can achieve:

- pre-processing time: $O(n+m)$ worst-case;
- space consumption : $O(n+m)$ worst-case;
- query time: $O(1+\frac{n}{m})$ expected.

As aforementioned, m controls the trade-off between the space consumption and the expected query time.

When $m=\Theta(n)$, the space consumption is $O(n)$ and the expected query time is $O(1)$.

Universal Hashing

$O(1)$ Worst-Case Query Time with $m = \Theta(n^2)$

Next, we show that with $m = \Theta(n^2)$, we can achieve $O(1)$ worst-case query time.

Denote by C_{all} the total number of pairs of distinct items in S that collide in T_h. By setting $m = n^2$, we have:

$$E\left[C_{all}\right] = \sum_{x,y \in S \wedge x \neq y} E\left[C_{x,y}\right] \leq \binom{n}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{2} \cdot \frac{1}{m} \leq \frac{1}{2}.$$

By Markov's Inequality,

$$\Pr[C_{all} \geq 1] \leq \Pr\big[C_{all} \geq 2 \cdot E[C_{all}]\big] \leq \frac{1}{2}.$$

$O(1)$ Worst-Case Query Time with $m = \Theta(n^2)$

Therefore, when $m = n^2$, $\Pr[C_{all} < 1] > \frac{1}{2}$.

Hence, in expectation, 2 trials of picking h from $\mathcal{H}$ suffice to make $C_{all} < 1$ happen, in which case, a *collision-free* hash table is constructed.

The construction is as follows:

- Uniformly at random pick h from a universal family $\mathcal{H}$;
- Construct T_h;
- If there is a collision, destroy T_h and start again from the first step.

Analysis

- pre-processing time: $O(n + m) = O(n^2)$ expected
- space consumption: $O(n + m) = O(n^2)$ worst-case
- query time: $O(1)$ worst-case

Universal Hashing

Final Remark

So far, we have two types of hashing schemes:

- Type 1: $m = \Theta(n)$.
  - $O(n)$ space with $O(1)$ expected query time;
- Type 2: $m = n^2$.
  - $O(n^2)$ space with $O(1)$ worst-case query time.

Next, we introduce Perfect Hashing, which combines these two types of hashing schemes and achieves:

- pre-processing time: $O(n)$ expected;
- space consumption: $O(n)$ worst-case;
- query time: $O(1)$ worst-case.

Perfect Hashing

A Two-Level Hashing Scheme

The First Level: h, with $m = n$

Recall that C_{all} is the total number of collision pairs in T_h, and we have:

$$E\left[C_{all}\right] = \sum_{x,y \in S \wedge x \neq y} E\left[C_{x,y}\right] \leq \binom{n}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{2} \cdot \frac{1}{m}.$$

When $m = n$, we have $E\left[C_{all}\right] \leq \frac{n}{2}$.

By Markov's Inequality, we have:

$$\Pr[C_{all} \geq n] \leq \Pr\big[C_{all} \geq 2 \cdot E[C_{all}]\big] \leq \frac{1}{2}.$$

Therefore,

$$\Pr[C_{all} < n] > \frac{1}{2}.$$

Perfect Hashing

A Two-Level Hashing Scheme

The First Level: h, with $m = n$

Construction algorithm:

- Uniformly at random pick h from a universal family $\mathcal{H}$;
- Construct T_h;
- If $C_{all} > n$, destroy T_h and start again from the first step.

In expectation, only 2 trials of picking h from $\mathcal{H}$ suffice.

Therefore, the expected construction time of the first level is $O(n)$.

Perfect Hashing

A Two-Level Hashing Scheme

After the first level hash table, T_h, is constructed, consider the linked list L_i stored at $T_h[i]$, where $i = 0, 1, \ldots, n-1$.

Let $n_i = |L_i|$.

Think: How many collision pairs of distinct elements in L_i?

Answer: There are $\binom{n_i}{2} = \Omega(n_i^2)$ collision pairs in L_i.

Observation: Summing over all the number of collision pairs in all L_i's, we have:

$$\sum_{i=0}^{n-1} \binom{n_i}{2} = C_{all} < n\,.$$

Perfect Hashing

A Two-Level Hashing Scheme

The Second Level: $g_i \in \mathcal{H}$, with $m = n_i^2$, for all i

According to the Type 2 Universal Hashing:

- a collision-free hash table on L_i's elements can be constructed in $O(n_i^2)$ expected time;
- the space consumption is $O(n_i^2)$.

Perfect Hashing

Space and Construction Time Analysis

The overall expected construction time of the two-level construction, is bounded by:

$$\begin{aligned} O(n) + \sum_{i=0}^{n-1} O(n_i^2) &= O(n) + O(\sum_{i=0}^{n-1} \binom{n_i}{2}) \\ &= O(n) + O(C_{all}) \\ &= O(n) + O(n) \\ &= O(n) \end{aligned}$$

The analysis for the overall space consumption is the same.

Perfect Hashing

Query Algorithm

Given a query element q, the query algorithm is as follows:

- Compute $i = h(q)$;
- Compute $k = g_i(q)$;
- Check $T_{g_i}[k]$ to decide whether q is in S:
  - If $T_{g_i}[k] = q$, then report $q \in S$. Otherwise, report $q \notin S$.

Since T_{g_i} is collision-free, $T_{g_i}[k]$ is either empty or stores only one element.

Think: Query time?

Answer: The query time is $O(1)$ worst-case.

Perfect Hashing

Summary

For a given static element set S, there exists a perfect hashing scheme with:

- pre-processing time: $O(n)$ expected;
- space consumption: $O(n)$ worst-case;
- query time: $O(1)$ worst-case.

A Universal Hashing Family

A Universal Hashing Family

Let p be a *prime* number such that $p > |\mathcal{U}|$.

Let a be an integer in $[p]^+ = \{1, 2, \ldots, p-1\}$.

Let b be an integer in $[p] = \{0, 1, 2, \ldots, p-1\}$.

Consider the hash function family

$$\mathcal{H} = \{h(x) = ((ax + b) \bmod p) \bmod m \mid a \in [p]^+, b \in [p]\}\,.$$

The hash function family $\mathcal{H}$ is two-universal.

We leave the proof of this claim as an exercise.