Hashing (Part 1)

CS 7800/4810 Lecture Notes

> *"The three most important data structures are hashing, hashing, hashing."* — Udi Manber

1 Introduction

Topics covered in this chapter:

- Universal and k-wise independent hashing
- Simple tabulation hashing
- Chaining and perfect hashing
- Linear probing
- Robin Hood hashing
- Cuckoo hashing

2 Hash Functions

A **hash function** maps keys from a universe to table slots:

$$h : \{0, 1, \ldots, u-1\} \to \{0, 1, \ldots, m-1\}$$

where $u = |U|$ is the size of the universe of keys and m is the table size.

2.1 Totally Random Hashing

A totally random hash function is the "gold standard" — it's what we'd have if we could pick a completely random mapping from keys to table slots.

Definition 2.1 (Totally Random Hash Function)**.** *A hash function $h : U \to \{0, 1, \ldots, m-1\}$ is **totally random** if:*

1. *For any key x and any slot t:* $\Pr[h(x) = t] = \frac{1}{m}$
2. *For any two distinct keys $x \neq y$: the values $h(x)$ and $h(y)$ are **independent***

This means every key lands uniformly at random in any slot, and knowing where one key lands tells you nothing about where any other key lands.

Why It's Ideal for Analysis

With totally random hashing, the balls-and-bins analysis applies directly:

- Expected number of keys per slot: $\frac{n}{m}$
- Maximum load in any bin: $\Theta\left(\frac{\log n}{\log \log n}\right)$ w.h.p.
- All the nice concentration bounds (Chernoff, etc.) apply cleanly

This is why CLRS uses the "simple uniform hashing assumption" — it makes the analysis tractable.

The Problem: Storage Cost

To specify a truly random function, you need to store where *every* key in the universe maps to.

How much space? For each of the $u = |U|$ keys, you need to store which of the m slots it maps to. That requires $\log_2 m$ bits per key, so:

$$\text{Total space} = u \cdot \lg m = \Theta(u \lg m) \text{ bits}$$

Example: If your keys are 64-bit integers ($u = 2^{64}$) and you have a table of size $m = 2^{20}$:

$$\text{Space} = 2^{64} \times 20 \text{ bits} \approx 10^{20} \text{ bits}$$

That's absurdly large — far bigger than any hash table you'd actually want to build!

Conceptual Picture

Think of a totally random hash function as a giant lookup table where each entry is chosen independently and uniformly at random from $\{0, 1, \ldots, m-1\}$:

Key	**Hash Value**
0	7
1	3
2	7
3	0
$\vdots$	$\vdots$
$u-1$	5

To evaluate $h(x)$, you just look up row x — but you need to store all u rows!

Why We Need Alternatives

In practice, we only insert $n \ll u$ keys into the table. We want:

1. **Small description:** $O(n)$ or $O(\text{poly} \log n)$ bits to describe h
2. **Fast evaluation:** $O(1)$ time to compute $h(x)$
3. **Good behavior:** Similar guarantees to totally random hashing

This is exactly what universal hashing and k-wise independence provide — they give us "good enough" randomness properties with small, efficiently computable hash functions.

When "Totally Random" Is Used

Despite being impractical to implement, totally random hashing is useful for:

1. **Analysis:** Prove bounds assuming totally random, then show weaker hash families achieve the same bounds
2. **Simulations:** When studying hash table behavior theoretically
3. **Lower bounds:** Show that even with perfect randomness, certain limits exist

For example, the $\Theta\left(\frac{\log n}{\log \log n}\right)$ maximum load bound holds even with totally random hashing — no hash function can do better in the worst case!

2.2 Universal Hashing

Definition 2.2 (Universal Hashing, Carter & Wegman, JCSS 1979)**.** *Choose h from family H with:*

$$\Pr_{h \in H}[h(x) = h(y)] = O\left(\frac{1}{m}\right) \quad \textit{for all } x \neq y \in U$$

*A family is **strongly universal** if the probability is exactly $\frac{1}{m}$.*

Example: Multiplicative Hashing (Universal)

$$h_a(x) = [(ax) \mod p] \mod m \quad \text{for } 0 < a < p$$

where $p > u = |U|$ is prime.

Construction details:

- **Universe:** Keys are from $U = \{0, 1, \ldots, u-1\}$
- **Prime p:** Choose any prime $p > u$. This ensures all keys are in $\{0, 1, \ldots, p-1\}$
- **Parameter a:** Choose uniformly at random from $\{1, 2, \ldots, p-1\}$
- **Why $a \neq 0$:** If $a = 0$, then $h(x) = 0$ for all x — every key collides!
- **Why $a < p$:** Working modulo p, values $\geq p$ are redundant

Why it works: For distinct $x \neq y$, the value $(ax) \mod p$ is uniformly distributed over $\{0, 1, \ldots, p-1\}$ as a varies. Since p is prime and $a \neq 0$, multiplication by a is a bijection on $\mathbb{Z}_p^*$.

The family: $H = \{h_a : 1 \leq a < p\}$ has $|H| = p - 1$ hash functions.

Example: Multiply-Shift [Dietzfelbinger et al., J. Alg. 1997]

$$h_a(x) = (ax) \gg (\lg u - \lg m)$$

where m and u are powers of 2 and $\gg$ denotes right bit-shift.

This extracts the higher-order bits after multiplication, which have better randomness properties than lower-order bits.

2.3 k-wise Independence

Definition 2.3 (k-wise Independence, Wegman & Carter, JCSS 1981)**.** *A family H of hash functions is k-**wise independent** if:*

$$\Pr_{h \in H}[h(x_1) = t_1 \text{ and } \ldots \text{ and } h(x_k) = t_k] = O\left(\frac{1}{m^k}\right)$$

for all distinct $x_1, x_2, \ldots, x_k \in U$.

Note: Pairwise independence ($k = 2$) is stronger than universal hashing.

Example: Pairwise Independent Hash Function

$$h_{a,b}(x) = [(ax + b) \mod p] \mod m \quad \text{for } 0 < a < p \text{ and } 0 \leq b < p$$

Construction details:

- **Prime p:** Choose any prime $p > u$ (same as before)
- **Parameter a:** Choose uniformly at random from $\{1, 2, \ldots, p-1\}$
- **Parameter b:** Choose uniformly at random from $\{0, 1, \ldots, p-1\}$
- **Why $a \neq 0$:** Ensures different keys map to different intermediate values
- **Why b can be 0:** The offset b just shifts all values; $b = 0$ is a valid shift

Why it's pairwise independent: For any two distinct keys $x \neq y$ and any two target values t_1, t_2, we need:

$$\Pr_{a,b}[h(x) = t_1 \text{ and } h(y) = t_2] = \frac{1}{m^2}$$

The key insight is that the system of linear equations:

$$ax + b \equiv r_1 \pmod{p}$$
$$ay + b \equiv r_2 \pmod{p}$$

has a *unique solution* (a, b) for any choice of r_1, r_2 (when $x \neq y$).

Proof sketch: Subtracting the equations gives $a(x - y) \equiv r_1 - r_2 \pmod{p}$. Since $x \neq y$ and p is prime, $(x - y)$ has a multiplicative inverse mod p, so a is uniquely determined. Then $b = r_1 - ax \mod p$ is also unique.

Since (a, b) is chosen uniformly from $(p-1) \times p$ possibilities, and each (r_1, r_2) pair corresponds to exactly one (a, b), the hash values are uniformly distributed.

The family: $H = \{h_{a,b} : 1 \leq a < p, 0 \leq b < p\}$ has $|H| = (p-1) \cdot p$ hash functions.

Example: k-wise Independent Hash Function

$$h_{a_0,\ldots,a_{k-1}}(x) = \left[\left(\sum_{i=0}^{k-1} a_i x^i\right) \mod p\right] \mod m$$

where $0 < a_{k-1} < p$ and $0 \leq a_i < p$ for $i < k - 1$.

Construction details:

- This is a random polynomial of degree $k-1$ over $\mathbb{Z}_p$
- **Parameters:** k coefficients $a_0, a_1, \ldots, a_{k-1}$
- **Why $a_{k-1} \neq 0$:** Ensures the polynomial has degree exactly $k-1$
- **Other coefficients:** Can be any value in $\{0, 1, \ldots, p-1\}$

Why it's k-wise independent: For any k distinct points $x_1, \ldots, x_k$, a degree $k-1$ polynomial is uniquely determined by its values at these points (polynomial interpolation). Thus the k hash values are uniformly and independently distributed.

Complexity: $O(k)$ space and $O(k)$ time to evaluate (using Horner's method).

2.4 Simple Tabulation Hashing

- View x as a vector of characters: $x_1, x_2, \ldots, x_c$
- Create a totally random hash table T_i on each character
- Requires $O(c \cdot u^{1/c})$ words of space

The hash function is:

$$h(x) = T_1(x_1) \oplus T_2(x_2) \oplus \cdots \oplus T_c(x_c)$$

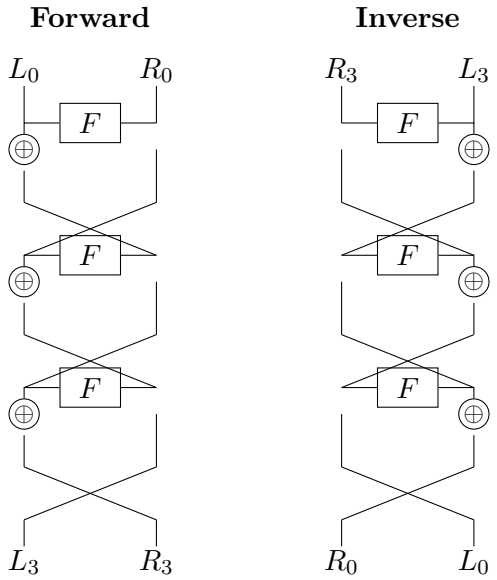
where $\oplus$ denotes XOR.

Properties:

- $O(c)$ time to compute
- 3-independent

2.5 Converting Non-Invertible to Invertible Hash Functions

Convert any n-bit to $2n$-bit hash function using a **Feistel Network** (also known as Luby-Rackoff block cipher).



Forward: Given input (L_0, R_0):

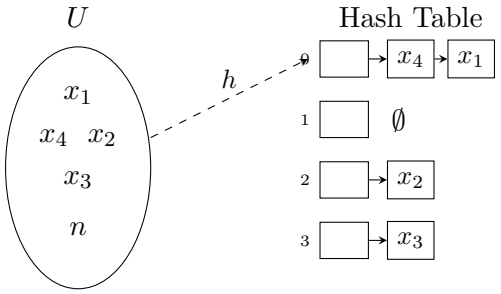
$$L_{i+1} = R_i$$
$$R_{i+1} = L_i \oplus F(R_i)$$

Inverse: Given (R_n, L_n), recover (L_0, R_0):

$$R_i = L_{i+1}$$
$$L_i = R_{i+1} \oplus F(L_{i+1})$$

3 Chaining

In chaining, each hash table slot contains a linked list of all elements that hash to that slot.



3.1 Expected Chain Length

$$E[C_t = \text{length of chain } t] = \sum_i \Pr\{h(x_i) = t\}$$

With universal hashing:

$$E[C_t] = O\left(\frac{n}{m}\right) = O(1) \quad \text{for } m = \Omega(n)$$

3.2 Variance of Chain Length

Step 1: Variance Formula

$$\text{Var}[C_t] = E[C_t^2] - E[C_t]^2$$

We already know $E[C_t] = \frac{n}{m}$, so $E[C_t]^2 = \frac{n^2}{m^2}$.
We need to compute $E[C_t^2]$.

Step 2: Define Indicator Variables

For each key x_i, define an indicator variable:

$$X_i = \begin{cases} 1 & \text{if } h(x_i) = t \\ 0 & \text{otherwise} \end{cases}$$

Then the chain length is:

$$C_t = \sum_{i=1}^{n} X_i$$

Step 3: Expand C_t^2

$$C_t^2 = \left(\sum_{i=1}^{n} X_i\right)^2 = \sum_{i=1}^{n}\sum_{j=1}^{n} X_i X_j$$

Split into diagonal terms ($i = j$) and off-diagonal terms ($i \neq j$):

$$C_t^2 = \sum_{i=1}^{n} X_i^2 + \sum_{i \neq j} X_i X_j$$

Step 4: Simplify X_i^2

Since $X_i \in \{0, 1\}$, we have $X_i^2 = X_i$.

$$E[X_i^2] = E[X_i] = \frac{1}{m}$$

So:

$$E\left[\sum_{i=1}^{n} X_i^2\right] = \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m}$$

Step 5: Handle Cross Terms $X_i X_j$

For $i \neq j$:

$$X_i X_j = \begin{cases} 1 & \text{if } h(x_i) = t \text{ AND } h(x_j) = t \\ 0 & \text{otherwise} \end{cases}$$

So:

$$E[X_i X_j] = \Pr\{h(x_i) = t \text{ and } h(x_j) = t\}$$

Step 6: Use Symmetry

Assuming h is "symmetric" (all slots are equally likely), we use the fact that:

$$E[C_t^2] = \frac{1}{m}\sum_{s=1}^{m} E[C_s^2]$$

Now, summing over all slots:

$$\sum_{s=1}^{m} E[C_s^2] = \sum_{s=1}^{m} E\left[\sum_{i,j} X_i^{(s)} X_j^{(s)}\right]$$

where $X_i^{(s)}$ indicates whether x_i hashes to slot s.
Rearranging:

$$\sum_{s}\sum_{i \neq j} E[X_i^{(s)} X_j^{(s)}] = \sum_{i \neq j}\sum_{s} \Pr\{h(x_i) = s \text{ and } h(x_j) = s\} = \sum_{i \neq j} \Pr\{h(x_i) = h(x_j)\}$$

Step 7: Apply Universal Hashing

With universal hashing, for any $i \neq j$:

$$\Pr\{h(x_i) = h(x_j)\} \leq \frac{c}{m}$$

for some constant c (typically $c = 1$ for strongly universal).
There are $n(n-1) < n^2$ pairs with $i \neq j$, so:

$$\sum_{i \neq j} \Pr\{h(x_i) = h(x_j)\} \leq n^2 \cdot \frac{c}{m}$$

Step 8: Compute $E[C_t^2]$

$$E[C_t^2] = \frac{1}{m}\left(n + n^2 \cdot O\left(\frac{1}{m}\right)\right) = \frac{n}{m} + O\left(\frac{n^2}{m^2}\right)$$

For $m = \Omega(n)$:

$$E[C_t^2] = O(1) + O(1) = O(1)$$

Step 9: Final Variance

$$\text{Var}[C_t] = E[C_t^2] - E[C_t]^2 = O(1) - O(1) = O(1)$$

Summary

Quantity	**Value**	**With $m = \Omega(n)$**
$E[C_t]$	$\frac{n}{m}$	$O(1)$
$E[C_t]^2$	$\frac{n^2}{m^2}$	$O(1)$
$E[C_t^2]$	$\frac{n}{m} + O\left(\frac{n^2}{m^2}\right)$	$O(1)$
$\text{Var}[C_t]$	$E[C_t^2] - E[C_t]^2$	$O(1)$

Key takeaway: Universal hashing gives us bounded variance, meaning chain lengths are concentrated around their expected value.

3.3 Maximum Chain Length with High Probability

Chernoff Bound

Let μ be the mean. For $c = 1 + \epsilon$:

$$\Pr\{C_t > c\mu\} \leq \frac{e^{(c-1)\mu}}{(c\mu)^{c\mu}}$$

This provides exponentially decreasing bounds on the probability of deviation of a random variable from its expected value.

Results:

- Totally random: $C_t = O\left(\frac{\lg n}{\lg \lg n}\right)$ w.h.p.

- $\Theta\left(\frac{\lg n}{\lg \lg n}\right)$-wise independence $\Rightarrow$ same bound
- Simple tabulation hashing $\Rightarrow$ same bound
- With "cache" of $O(\lg n)$ items, totally random gives $O(1)$ amortized w.h.p. [Pătraşcu, blog 2011]
- $\Theta(\lg n)$ keys collide with "batch" of $\lg n$ operations w.h.p.
- For $\mu = \lg n$, $c = 2$: $\Pr \leq \frac{e^{\lg n}}{(2 \lg n)^{2 \lg n}} < \frac{1}{n^c}$