Hashing and Bloom Filters

Algorithmics study and revision notes

Jaak Vilo

2026-08-22

Contents

1 Why hashing?

A **dictionary** stores key-value associations and supports search, insertion, and deletion by key. Direct addressing is ideal when keys already lie in a small universe: store the value for key x in array cell x. When the possible key universe is enormous compared with the number of stored keys, that array wastes space.

Hashing maps a large universe into a manageable table. The price is unavoidable collisions: different keys can map to the same cell. A hash table is correct because it resolves collisions and checks actual key equality, not because its hash function somehow prevents all collisions.

2 Learning goals

After studying this note, you should be able to:

- separate the dictionary interface, hash code, table index, and collision policy;
- trace chaining and open addressing, including deletion;
- derive expected collision and empty-cell counts under a uniform model;
- explain load factor, resizing, and expected versus amortised time;
- instantiate a universal hash family and state its guarantee;
- derive Bloom-filter false-positive probability and choose its parameters; and
- distinguish table hashing, cryptographic hashing, password hashing, and Zobrist hashing.

3 Prerequisites

- Arrays, linked structures, modular arithmetic, and geometric resizing.
- Elementary probability: linearity of expectation and the approximation $(1-x)^r \approx e^{-rx}$ for small x.
- The distinction between worst-case, expected, and amortised bounds.

4 The dictionary model

A dictionary maintains a set of entries $(key, value)$, normally with unique keys. Its abstract operations are:

```
insert_or_replace(D, key, value)
search(D, key)
delete(D, key)
```

The model should specify key equality and mutability. If a key changes after insertion while its stored hash position does not, later search may fail.

A **hash code** maps a key to a large integer, often a machine word. A separate compression step maps that integer to a table index in $\{0, \dots, m-1\}$. For example,

$$index(key) = hashcode(key) \bmod m.$$

This separation matters when resizing: changing m changes table indices even when hash codes remain fixed.

5 What a table hash function must do

Equal keys must produce equal hash codes. For expected performance, the indexed values of the anticipated keys should be spread across the table without strong patterns. Fast mixing is useful, but it is not the same goal as cryptographic preimage or collision resistance.

For composite objects, combine every equality-relevant field in a defined order. A string hash, for example, can update an integer accumulator once per character and then apply a table-specific compression or

universal-hashing step. Test a proposed function on representative and adversarial-looking data; visual uniformity alone is not a proof.

6 Separate chaining

Each table cell stores a **bucket** containing all entries whose index equals that cell. A bucket may be a list, compact array, or another small dictionary.

With n entries and m buckets, the **load factor** is

$$\alpha = \frac{n}{m}.$$

Under simple uniform hashing, a bucket has expected length α, and search, insertion, and deletion take expected $O(1 + \alpha)$ time. Chaining permits $\alpha > 1$ and makes deletion direct, but pointer-heavy buckets can have poor locality.

Correct search first selects the bucket, then compares the query key with each candidate's actual key. Comparing hash codes alone is incorrect because unequal keys can share a code.

7 Open addressing

Open addressing stores every entry directly in the table. For key x, a probe sequence

$$h(x, 0), h(x, 1), \ldots, h(x, m - 1)$$

specifies cells to inspect until the key or an available cell is found.

- **Linear probing:** $h(x, i) = (h_1(x) + i) \bmod m$. It has excellent locality but creates primary clusters.
- **Quadratic probing:** increments grow quadratically. It reduces primary clustering, but parameters and table size must be chosen so enough cells are reachable.
- **Double hashing:** $h(x, i) = (h_1(x) + i\, h_2(x)) \bmod m$. To visit every cell, the step $h_2(x)$ must be relatively prime to m; a prime table size and a nonzero step smaller than m are a common design.

Search must follow exactly the insertion probe sequence. Deleting an item by marking its cell **empty** can break a later search whose probe path crosses that cell. Use a **tombstone** that search passes but insertion may reuse, or rebuild the affected table.

Open addressing requires $\alpha < 1$. Under the idealised uniform-probing model, expected probes for an unsuccessful search are at most

$$\frac{1}{1 - \alpha},$$

and for a successful search at most

$$\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}.$$

Real linear probing has correlated probes, so these formulas are a model, not a guarantee. Performance rises sharply as the table becomes full.

8 Resizing and amortised time

When load passes a chosen threshold, allocate a geometrically larger table and reinsert every live entry. Entries must be reinserted because their indices depend on the new size; copying cells to equal indices is not enough.

A resize costs $\Theta(n)$, but geometric growth makes a long sequence of insertions amortised $O(1)$ under the hashing assumptions. Each entry can be charged a small amount on insertion to pay for the occasional rebuild. Tombstones may also trigger a same-size rebuild even when the live load is moderate.

Always distinguish the two probability layers:

- **expected:** over a random hash choice or key-distribution model;
- **amortised:** over an operation sequence including resizing.

An ordinary hash table still has $\Theta(n)$ worst-case search if many keys land together.

9 Collision and occupancy estimates

Suppose n distinct keys are hashed independently and uniformly into m cells. Each unordered key pair collides with probability $1/m$. By linearity of expectation,

$$E[\text{colliding pairs}] = \binom{n}{2}\frac{1}{m} = \frac{n(n-1)}{2m}.$$

For $n \le m$, the exact no-collision probability is

$$\Pr(\text{no collision}) = \prod_{i=0}^{n-1}\left(1 - \frac{i}{m}\right).$$

When n is small relative to m, taking logarithms and retaining the leading term gives the birthday approximation

$$\Pr(\text{no collision}) \approx \exp\left(-\frac{n(n-1)}{2m}\right).$$

The probability of at least one collision reaches about one half near

$$n \approx \sqrt{2m \ln 2}.$$

For any particular cell, the probability of remaining empty is $(1 - 1/m)^n$. Hence

$$E[\text{empty cells}] = m\left(1 - \frac{1}{m}\right)^n \approx me^{-n/m}.$$

Define the reported collision metric. “Colliding pairs,” “keys beyond the first in an occupied cell,” and “number of occupied cells with load at least two” are different quantities.

10 Universal hashing

A fixed deterministic hash can perform badly on a carefully selected key set. **Universal hashing** chooses one function randomly from a family after the key set is fixed. A family $\mathcal{H}$ mapping universe $\mathcal{U}$ to m cells is universal if, for every distinct x, y,

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \le \frac{1}{m}.$$

A standard construction chooses a prime p larger than every encoded key, chooses a uniformly from $\{1, \dots, p-1\}$ and b uniformly from $\{0, \dots, p-1\}$, and uses

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod m.$$

The random parameters are selected once per table and stored with it. For two distinct keys, the affine map modulo p randomises their relative residues sufficiently to bound their chance of landing in the same final bucket. The guarantee is about collision probability over the function choice; it does not say one realised table has no collisions.

More explicitly, for distinct x, y, the pair

$$((ax + b) \bmod p, (ay + b) \bmod p)$$

is uniform over ordered pairs of distinct residues. After fixing the first residue, at most $\lceil p/m \rceil - 1 \leq (p-1)/m$ of the other $p-1$ residues have the same remainder modulo m. Hence the collision probability is at most $1/m$.

11 Bloom filters

A Bloom filter represents approximate membership using an m-bit array and k hash-derived positions.

```
insert(x):
    for each position h_i(x): set bit h_i(x)

possibly_contains(x):
    return every bit h_i(x) is 1
```

If any queried bit is zero, the key is **definitely absent**. If all are one, it is **possibly present**. With consistent hashing and no unsupported deletion, inserted keys have no false negatives; unrelated keys can be false positives.

Under the standard model in which the kn selected positions are independent and uniform, a particular bit remains zero with probability

$$\left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}.$$

Under the usual independence approximation, all k queried positions are one with probability

$$p \approx \left(1 - e^{-kn/m}\right)^k.$$

For fixed bits per item m/n, this expression is minimised near

$$k = \frac{m}{n} \ln 2.$$

With ten bits per item, $k \approx 6.93$, so seven hashes are natural:

$$p \approx (1 - e^{-0.7})^7 \approx 0.0082.$$

This is a model prediction. Correlated or poorly distributed hashes can produce worse results. Compare measured false positives against the prediction on queries known not to be in the inserted set.

Practical filters often derive k positions from two base hashes rather than compute k unrelated hashes. This reduces hashing cost and can work well, but the independence calculation remains a model to validate empirically for the chosen scheme and data.

A standard Bloom filter cannot safely delete: clearing a bit shared with another item may create a false negative. A counting Bloom filter replaces bits with small counters, at greater space cost.

12 Security hashing is a different tool

These concepts share a name but solve different problems:

- A **table hash** is designed for fast distribution and lookup.
- A **cryptographic hash** is designed to resist preimages and deliberate collisions, but is usually intentionally too fast for password storage.
- A **keyed hash** can prevent an attacker from predicting table collisions when public inputs are adversarial.
- A **password-hashing or key-derivation function** is deliberately slow and often memory-hard.

A password salt is a public, unique random value stored with the password verifier. It prevents identical passwords from sharing one stored result and defeats one precomputed table across many accounts. It does not turn a fast general hash into a suitable password-hashing scheme.

Using a cryptographic digest and then reducing it modulo m can distribute table indices well, but usually costs more CPU than a noncryptographic table hash. Performance and threat model should decide whether that cost is justified.

13 Zobrist hashing

Search programs repeatedly modify structured states such as game boards. **Zobrist hashing** assigns a random machine word $R[position, piece]$ to every position-piece combination and XORs the words for all occupied positions:

$$H(state) = \bigoplus_{(pos,piece)\ \text{occupied}} R[pos, piece].$$

For a Tic-Tac-Toe move placing X in cell 4, update in constant time with

$$H \leftarrow H \oplus R[4, X].$$

Moving or replacing a piece XORs out its old contribution and XORs in its new one. The hash is an efficient fingerprint for transposition tables, not a proof of state equality; store enough state information to detect the rare collision.

14 Enrichment: static and similarity hashing

A **perfect hash function** has no collisions on one fixed key set. Two-level randomized schemes can build a static dictionary with expected linear space and construction time and worst-case constant lookup. A **minimal perfect hash** maps n fixed keys into exactly n indices, but normally needs extra storage to verify whether an arbitrary query belongs to the original set.

Locality-sensitive hashing deliberately gives similar objects an elevated chance of colliding. That goal is nearly the opposite of ordinary table hashing and is useful for approximate similarity search. It requires a defined distance or similarity family and its own probability analysis.

15 Worked trace: tombstones and collision estimates

Let $m = 7$, use $h(x) = x \bmod 7$, and resolve collisions by linear probing.

1. Insert 10 at cell 3.
2. Key 17 also starts at 3, so place it at 4.
3. Key 24 starts at 3, passes occupied cells 3 and 4, and enters cell 5.
4. Delete 17 by placing a tombstone at 4.
5. Searching for 24 must pass cell 4 and succeeds at 5. If cell 4 had been marked empty, search would incorrectly stop.

For a separate experiment with $m = 100\,000$ and $n = 1\,000$ uniformly hashed keys,

$$E[\text{colliding pairs}] = \frac{1000 \cdot 999}{2 \cdot 100000} \approx 4.995.$$

The birthday estimate predicts only $e^{-4.995} \approx 0.0068$ probability of no collision. A collision is therefore unsurprising even though only one percent of the table's capacity is used.

16 Complexity and trade-offs

Structure	Lookup	Insert	Delete	Main assumptions
Direct-address table	$O(1)$ worst case	$O(1)$	$O(1)$	small integer universe
Chained hash table	expected $O(1+\alpha)$	expected $O(1)$ amortised	expected $O(1+\alpha)$	distributed hashes and resizing
Open addressing	expected $O(1)$ at controlled α	expected amortised $O(1)$	expected $O(1)$	valid probe sequence and tombstones
Balanced search tree	$O(\log n)$ worst case	$O(\log n)$	$O(\log n)$	comparable keys; preserves order
Bloom filter	$O(k)$	$O(k)$	unsupported normally	approximate membership only

Hash tables do not preserve order. Use a search tree or another ordered index for predecessor, successor, sorted iteration, and range queries.

17 Connections

- Probability and asymptotic analysis turn occupancy assumptions into measurable predictions.
- Geometric resizing uses the same amortised reasoning as dynamic arrays.
- Bloom filters are compact prefilters for databases, storage systems, and web caches.
- Zobrist hashing supports duplicate-state detection in heuristic search.
- Hashing can memoise dynamic-programming or recursive states when the state space is sparse.

18 Common mistakes

- Treating a collision as an error rather than a normal event.
- Comparing only hash codes instead of actual keys.
- Clearing an open-addressing cell and breaking a later probe path.
- Confusing expected constant time with worst-case constant time.
- Reporting "collisions" without defining the counted quantity.
- Treating "possibly present" from a Bloom filter as proof of membership.
- Using an unsalted fast hash as password storage.
- Forgetting that a Zobrist fingerprint can collide.

19 Self-check

1. Why must a resized table reinsert rather than merely copy its cells?
2. Trace insertion and deletion of three colliding keys under quadratic probing.
3. Estimate the first-collision scale for a table with one million cells.

4. Derive the expected number of empty cells from one cell's empty probability.
5. What randomness appears in the universal-hashing guarantee?
6. Which Bloom-filter answer is certain, and which is probabilistic?
7. Explain the distinct roles of a password salt and a slow password-hashing function.
8. Why is XOR useful for Zobrist updates?

20 Revision summary

- Hashing implements a dictionary by combining a mapping, collision policy, equality check, and resizing policy.
- Chaining and open addressing respond differently to load and deletion.
- Collision and occupancy counts follow from explicit probability models.
- Universal hashing randomises the function choice against fixed bad key sets.
- Bloom filters exchange controlled false positives for compact negative filtering.
- Cryptographic, password, and Zobrist hashing have goals distinct from ordinary table lookup.

21 Implementations and hands-on exploration

- Python's built-in `dict` is the natural dictionary baseline and preserves insertion order. Its public interface does not expose probe counts or load thresholds, so use a teaching implementation when those internal measurements are the experiment.
- uthash adds macro-based hash tables to C by embedding a hash handle in each user structure. Include that handle, bucket table, and allocator overhead when comparing memory with a flat array implementation.
- Abseil's C++ `flat_hash_map` is a production Swiss-table implementation optimised for locality. Rehashing invalidates iterators, references, and pointers to elements; use `node_hash_map` when pointer stability is required.
- Rust's standard `HashMap` uses a randomly seeded hasher and a SwissTable-derived layout. Iteration order is unspecified, and replacing the default hasher trades security and speed properties that should be stated.
- Guava's Java `BloomFilter` exposes expected insertions and target false-positive probability directly. Inserting far beyond that estimate saturates the filter and sharply worsens its false-positive rate; ordinary deletion is unsupported.

21.1 Small experiments

1. For several pairs (n, m), choose a fresh universal hash function for each trial, insert fixed distinct integers, and record colliding pairs and empty cells. Compare the sample means with $n(n-1)/(2m)$ and $m(1-1/m)^n$.
2. Build Bloom filters at fixed bits per item while varying k around $(m/n)\ln 2$. Query a disjoint held-out set, plot measured versus predicted false-positive rates, verify that inserted items have no false negatives, and then deliberately overfill one filter.

22 Sources and further study

- Hashing lecture deck, Jaak Vilo.
- (Cormen et al. 2022), Chapter 11.
- NIST Digital Identity Guidelines: Authentication and Authenticator Management, National Institute of Standards and Technology.

References

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. 4th ed. MIT Press.