Week 3 — Hash Tables

Félix Chavelli felix.chavelli@inria.fr

March 4, 2026 · Semester 2

PSL

Today's Agenda

Motivation & Overview

Direct-Address Tables (CLRS 11.1)

Hash Tables with Chaining (CLRS 11.2)

Hash Functions (CLRS 11.3)

Open Addressing (CLRS 11.4)

Perfect Hashing (Overview)

Practice: Python & C++

Summary & Takeaways

Part 1

Motivation & Overview

The Dictionary Problem

- Many applications need a **dynamic set** supporting:
  - $\textsc{Insert}(S, x)$
  - $\textsc{Search}(S, k)$
  - $\textsc{Delete}(S, x)$
- Symbol tables, caches, databases, deduplication...
- **Goal:** all three operations in $\mathcal{O}(1)$ *average* time

	Search	Insert	Delete
Sorted arr.	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Linked list	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
BST (bal.)	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Hash table	$\mathcal{O}(1)$*	$\mathcal{O}(1)$*	$\mathcal{O}(1)$*

* average, under good hashing

Key Idea

Hash tables achieve $\mathcal{O}(1)$ average-case dictionary operations by computing array indices from keys via a **hash function**.

Hash Tables Everywhere

- **Languages:** Python `dict`/`set`, C++ `unordered_map`/`unordered_set`, Java `HashMap`
- **Databases:** hash indexes, hash joins
- **Caching:** memoization, web caches, CDN routing
- **Security:** password storage, digital signatures, blockchain (Merkle trees)
- **AI / ML:** feature hashing ("hashing trick"), embedding tables, deduplication of training data
- **Distributed systems:** consistent hashing (Chord, DynamoDB)
- **Compilers:** symbol tables for identifiers

→ Today: the theory and practice behind all of this

S1 Recap & What's New

Already seen in S1:

- Hash function: maps keys to indices
- Collisions happen; chaining & open addressing exist
- $\Theta(1)$ amortized for `dict`/`set`
- Python `set`/`dict`, C++ `unordered_map`/`unordered_set`

New today (CLRS 11):

- Direct-address tables (11.1)
- **Formal analysis** of chaining: load factor, Theorems 11.1–11.2 (11.2)
- Hash functions: division, multiplication, **universal hashing** (11.3)
- Open addressing in depth: linear probing, double hashing, analysis (11.4)
- Perfect hashing overview (11.5)

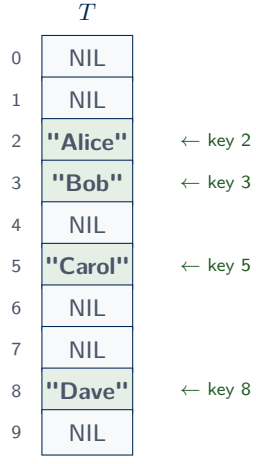
Part 2

Direct-Address Tables (CLRS 11.1)

Direct-Address Tables — Idea

Direct-Address Table

If the universe of keys is $U = \{0, 1, \ldots, m-1\}$ with m not too large, use an array $T[0\,..\,m-1]$ where slot k stores the element with key k.



Operations — all $\mathcal{O}(1)$:

- $\textsc{Search}(T, k)$: return $T[k]$
- $\textsc{Insert}(T, x)$: $T[x.key] = x$
- $\textsc{Delete}(T, x)$: $T[x.key] = \textsc{nil}$

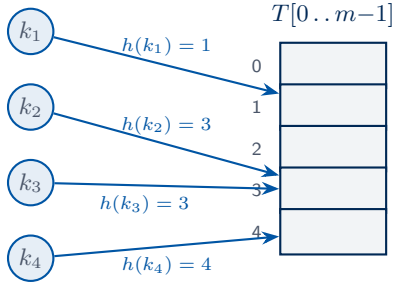
Limitation

Requires $\Theta(|U|)$ memory. If $|U|$ is large (e.g. 64-bit integers), this is impractical!

From Direct Addressing to Hashing

Key Idea

When $|U| \gg n$ (number of keys stored), allocate a table of size $m = \Theta(n)$ and use a **hash function** $h : U \to \{0, 1, \ldots, m-1\}$ to map keys to slots.



- Space: $\Theta(m) = \Theta(n)$
- Problem: k_2 and k_3 map to the same slot
- This is a collision
- Since $|U| > m$, collisions are **unavoidable** (pigeonhole principle)

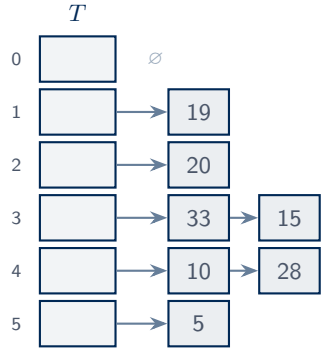
Part 3

Hash Tables with Chaining (CLRS 11.2)

Collision Resolution by Chaining

Chaining

Each slot $T[j]$ stores a (linked) list of all elements that hash to slot j.



$h(k) = k \bmod 6$; keys: 5, 28, 19, 15, 20, 33, 10

Operations:

- $\textsc{Insert}(T, x)$: prepend to list at $T[h(x.key)]$
  $\Rightarrow \mathcal{O}(1)$ worst case
- $\textsc{Search}(T, k)$: search list at $T[h(k)]$
  $\Rightarrow \mathcal{O}(|\text{list}|)$ worst case
- $\textsc{Delete}(T, x)$: remove x from its list (doubly linked $\Rightarrow \mathcal{O}(1)$)

Worst case: all n keys in one slot $\Rightarrow \Theta(n)$

Load Factor & Average-Case Analysis

Load Factor

For a hash table with m slots storing n elements: $\alpha = n/m$ is the **load factor** $=$ average $\#$ elements per chain. We assume **independent uniform hashing**.

Theorem 11.1 (CLRS)

An **unsuccessful search** takes $\Theta(1+\alpha)$ time on average.

Theorem 11.2 (CLRS)

A **successful search** takes $\Theta(1+\alpha)$ time on average.

Key Idea

If $n = \mathcal{O}(m)$ then $\alpha = \mathcal{O}(1)$, so **all dictionary operations take $\mathcal{O}(1)$ time on average**.

Proof Sketch — Unsuccessful Search

- Key k not in the table hashes to a uniformly random slot $h(k)$
- Expected length of list $T[h(k)]$ is $\mathrm{E}[n_{h(k)}] = \alpha = n/m$
- We scan the *entire* list $\Rightarrow$ expected elements examined $= \alpha$
- Total time: $\Theta(1 + \alpha)$ (1 for computing $h(k)$)

Successful search (proof idea):

- Element x_i (inserted i-th) has, on average, $(n - i)/m$ elements inserted after it that collide with it
- Average over all n elements:

$$\frac{1}{n} \sum_{i=1}^{n} \left(1 + \frac{n-i}{m}\right) = 1 + \frac{n-1}{2m} = 1 + \frac{\alpha}{2} - \frac{1}{2m} = \Theta(1 + \alpha)$$

See CLRS pp. 278–280 for the full proof using indicator random variables.

Practical Implications

α	Unsuccessful	Successful	Interpretation
0.5	1.5 probes	1.25 probes	Table half-full
0.75	1.75 probes	1.375 probes	Typical load
1.0	2.0 probes	1.5 probes	Full (1 elem/slot avg)
2.0	3.0 probes	2.0 probes	Overloaded

Key Idea

In practice, keep $\alpha \leqslant 0.75$ (Python uses $\approx 2/3$). **Resize** (double the table, rehash) when load factor exceeds threshold. Amortized cost of resizing: $\mathcal{O}(1)$ per insertion.

Part 4

Hash Functions (CLRS 11.3)

What Makes a Good Hash Function?

- **Deterministic:** same key $\Rightarrow$ same hash value (always)
- **Uniform:** each slot equally likely for a "random" key
- **Independent:** hash of k_1 tells nothing about hash of k_2
- **Fast:** $\mathcal{O}(1)$ to compute

Independent Uniform Hashing (ideal)

Each key k maps to a slot $h(k)$ chosen uniformly at random from $\{0, \ldots, m-1\}$, independently of all other keys.

This is a *theoretical ideal* (a "random oracle") — not implementable in practice, but a useful analysis tool.

Two practical approaches:

1. **Static hashing** (division, multiplication) — simple but no guarantee
2. **Random hashing** (universal families) — provable guarantees

The Division Method

Division Method

$$h(k) = k \bmod m$$

Example: $m = 12$, $k = 100 \Rightarrow h(k) = 4$

- Very fast: single modulo operation
- **Choice of m matters!**
  - Avoid $m = 2^p$ (only looks at low-order p bits)
  - Avoid m close to a power of 2
  - Best: m = prime, not too close to 2^p

Pitfall

If keys have patterns (e.g. all even), a bad m amplifies collisions.

Example: $m = 100$, keys are student IDs ending in same digits $\Rightarrow$ many collisions!

The Multiplication Method

Multiplication Method

Choose constant A with $0 < A < 1$:

$$h(k) = \lfloor m \cdot (kA \bmod 1) \rfloor$$

where $kA \bmod 1 = kA - \lfloor kA \rfloor$ is the fractional part.

- Value of m is **not critical** (works for any m)
- Knuth suggests $A \approx (\sqrt{5} - 1)/2 \approx 0.6180\ldots$ (golden ratio)

Multiply-shift variant ($m = 2^\ell$, word size w):

- Choose fixed w-bit odd integer a
- $h_a(k) = (k \cdot a \bmod 2^w) \gg (w - \ell)$
- Only 3 machine instructions: multiply, mod, shift
- Fast, but no *guarantee* of good average-case performance

Example: $k = 123456$, $w = 32$, $\ell = 14$, $a = 2654435769 \Rightarrow h_a(k) = 67$

Why Static Hashing is Not Enough

Adversarial inputs

For *any* fixed hash function h, an adversary can choose n keys that **all hash to the same slot** $\Rightarrow \Theta(n)$ search time.

Real-world attacks:

- HashDoS (2011): crafted HTTP POST parameters causing $\mathcal{O}(n^2)$ processing in web servers (Python, Java, PHP, Ruby...)
- Solution: **randomize** the hash function at program startup

Key Idea

Random hashing: choose h at random from a *family* $\mathcal{H}$ of hash functions, independently of the keys. No single input can always trigger worst-case behavior.

Universal Hashing

Universal Hash Family

A family $\mathcal{H}$ of hash functions $h : U \to \{0, \ldots, m-1\}$ is **universal** if for every pair of distinct keys $k_1 \neq k_2$:

$$\Pr_{h \in \mathcal{H}} \big[\, h(k_1) = h(k_2) \,\big] \leqslant \frac{1}{m}$$

- Collision probability $\leqslant 1/m$ = same as truly random hashing
- But we only need to store *two parameters*, not a full random oracle

Corollary 11.3 (CLRS)

Using **universal hashing** with chaining in a table of m slots, any sequence of s operations containing $n = \mathcal{O}(m)$ insertions takes $\Theta(s)$ expected time.

$\Rightarrow$ Each operation is $\mathcal{O}(1)$ expected, **regardless of the input**.

A Universal Family: $\mathcal{H}_{pm}$

Carter–Wegman family (1979)

Choose a prime $p \geqslant |U|$ and table size m. For $a \in \{1, \ldots, p-1\}$, $b \in \{0, \ldots, p-1\}$:

$$h_{a,b}(k) = \big((ak + b) \bmod p\big) \bmod m$$

Family: $\mathcal{H}_{pm} = \{h_{a,b} : a \in \mathbb{Z}_p^*,\ b \in \mathbb{Z}_p\}$ $\quad (p(p-1)$ functions).

Example: $p = 17$, $m = 6$

$$h_{3,4}(8) = ((3{\cdot}8 + 4) \bmod 17) \bmod 6 = 5$$

Proof idea (Thm 11.4):

- Distinct $k_1, k_2 \Rightarrow$ distinct $r_1, r_2 \pmod p$ (since p prime, $a \neq 0$)
- $(a, b) \leftrightarrow (r_1, r_2)$: bijection
- Collision iff $r_1 \equiv r_2 \pmod m$: $\leqslant \frac{p-1}{m}$ of $p-1$ choices

Universal Hashing in Practice

Multiply-shift (Thm 11.5):

- $m = 2^\ell$; pick random *odd* a
- $h_a(k) = (ka \bmod 2^w) \gg (w - \ell)$
- $2/m$-universal (slightly weaker)
- **Fastest in practice**: 3 instructions

Python's approach:

- `hash()` + random salt (`PYTHONHASHSEED`)
- SipHash since Python 3.4 (cryptographic quality)

Hashing non-integer keys:

- Strings: treat as radix-128 number, apply hash
- Vectors: $h_b(\langle a_0, \ldots, a_{d-1} \rangle) = \sum a_j b^j \bmod p$
- Cryptographic: SHA-256, SipHash, AES-based

Key Idea

In practice, use your language's built-in hash. Understand the theory to know *why* it works and to diagnose pathological cases.

Part 5

Open Addressing (CLRS 11.4)

Open Addressing — Idea

Open Addressing

All elements stored directly in the table (no linked lists). On collision, **probe** successive slots until an empty one is found.

- Hash function: $h : U \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$
- Probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ = **permutation** of $\{0, \ldots, m-1\}$
- Constraint: $\alpha \leqslant 1$ (table can fill up!)

Insertion:

```
1: i ← 0
2: repeat
3:     q ← h(k, i)
4:     if T[q] = NIL then
5:         T[q] ← k; return q
6:     end if
7:     i ← i + 1
8: until i = m
9: error "overflow"
```

Search:

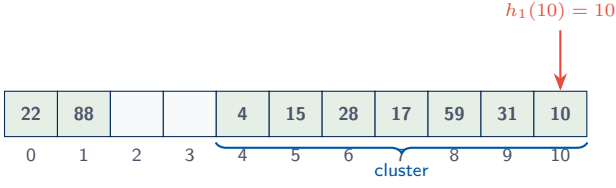
```
1: i ← 0
2: repeat
3:     q ← h(k, i)
4:     if T[q] = k then
5:         return q
6:     end if
7:     i ← i + 1
8: until T[q] = NIL or i = m
9: return NIL
```

Linear Probing

Linear Probing

$$h(k,i) = (h_1(k) + i) \bmod m$$

Simplest open addressing: try slots in sequence from $h_1(k)$.



$h_1(k) = k \bmod 11$ keys: 10, 22, 31, 4, 15, 28, 17, 88, 59

Primary clustering:

- Long runs of occupied slots build up
- Slot after cluster of size i: filled w.p. $(i+1)/m$
- Clusters grow and merge ⇒ degraded performance

✔ **Cache-friendly:** sequential access ⇒ fast on modern CPUs

Double Hashing

Double Hashing

Uses two auxiliary hash functions:

$$h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$$

Step size $h_2(k)$ varies per key $\Rightarrow$ breaks clustering.

Example: $m = 13$

$$h_1(k) = k \bmod 13$$
$$h_2(k) = 1 + (k \bmod 11)$$

Insert $k = 14$:

- $h(14, 0) = 1$ (occupied)
- $h(14, 1) = 4$ (occupied)
- $h(14, 2) = 7$ (occupied)
- $h(14, 3) = \mathbf{10}$ ✔ empty

Requirements:

- $h_2(k)$ must be **coprime** to m
- Easy if m prime:
  $h_2(k) = 1 + (k \bmod m')$
- Or $m = 2^p$ and h_2 always odd

- $\Theta(m^2)$ distinct probe sequences
- Close to ideal (uniform permutation hashing)

Analysis of Open Addressing

Assume **independent uniform permutation hashing** and $\alpha < 1$, no deletions.

Theorem 11.6 — Unsuccessful search

Expected number of probes $\leqslant \dfrac{1}{1-\alpha}$

Theorem 11.8 — Successful search

Expected number of probes $\leqslant \dfrac{1}{\alpha} \ln \dfrac{1}{1-\alpha}$

α	**Unsuccessful** $\leqslant 1/(1-\alpha)$	**Successful** $\leqslant \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
0.50	2.00	1.39
0.75	4.00	1.85
0.90	10.00	2.56
0.95	20.00	3.15

Proof Sketch — Unsuccessful Search (Open Addressing)

- Let A_i = event that probe i hits an occupied slot
- $\Pr\{X \geqslant i\} = \Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}$
- Under independent uniform permutation hashing:

$$\Pr\{A_1\} = \frac{n}{m}, \quad \Pr\{A_2 \mid A_1\} = \frac{n-1}{m-1}, \quad \dots$$

- Since $\frac{n-j}{m-j} \leqslant \frac{n}{m} = \alpha$ for $0 \leqslant j < m$:

$$\Pr\{X \geqslant i\} \leqslant \alpha^{i-1}$$

- Therefore:

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr\{X \geqslant i\} \leqslant \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

Intuition: geometric series — probability $\approx \alpha$ of needing each extra probe.

Deletion in Open Addressing

Deletion is tricky!

Cannot simply set a deleted slot to NIL — it would break search chains for keys that probed past that slot.

Two solutions:

1. **Tombstones (Deleted marker):**
   - Insert treats DELETED as empty, Search skips over it
   - Downside: search time no longer depends only on α
2. **Linear probing with shift-back deletion** (CLRS 11.5.1):
   - After deleting slot q, scan forward and shift keys back if needed
   - Works because linear probe sequences all follow $+1$ pattern
   - Clean: no tombstones, maintains performance properties

This is why **chaining** is often preferred when deletions are frequent.

Chaining vs. Open Addressing — Summary

	Chaining	Open Addressing
Extra memory	Pointers in lists	None (all in table)
Load factor	Can exceed 1	Must have $\alpha < 1$
Worst-case search	$\Theta(n)$	$\Theta(n)$
Average search	$\Theta(1+\alpha)$	$\leqslant \frac{1}{1-\alpha}$
Deletion	Easy (doubly linked)	Tricky (tombstones)
Cache behavior	Poor (pointer chasing)	Good (sequential)
Clustering	No	Yes (linear probing)

Key Idea

Modern high-performance implementations often use **open addressing** (or hybrid schemes) for cache efficiency. Python `dict` uses open addressing with perturbation; C++ `std::unordered_map` uses chaining by default.

Part 6

Perfect Hashing (Overview)

Perfect Hashing — $\mathcal{O}(1)$ Worst-Case Search

Perfect Hashing

For a **static** set of n keys (no insertions/deletions), we can build a hash table with **zero collisions** and $\mathcal{O}(n)$ space. $\Rightarrow \mathcal{O}(1)$ **worst-case** search time.

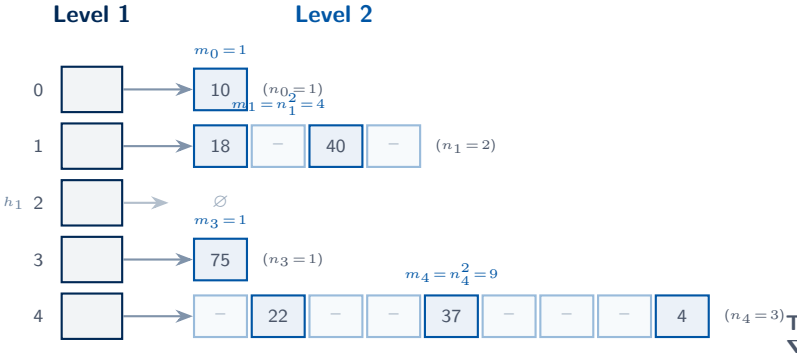
Two-level scheme (Fredman, Komlós, Szemerédi 1984):

1. **Level 1:** hash n keys into m slots using a universal family
2. **Level 2:** for each slot j with n_j collisions, build a *secondary* hash table of size n_j^2
   - n_j^2 slots $\Rightarrow$ zero collisions w.h.p. by birthday paradox
   - If collisions occur, re-pick the level-2 hash function until collision-free

Space Bound

With universal hashing at both levels, expected total space is $\mathcal{O}(n)$. Key insight: $\sum_{j=0}^{m-1} n_j^2 = \mathcal{O}(n)$ when $m = \Theta(n)$.

Perfect Hashing — Illustration



How to read this:

1. h_1 distributes $n = 7$ keys into $m = 5$ level-1 slots
2. Slot j receives n_j keys
3. Each level-2 table has $m_j = n_j^2$ slots and its own hash function h_j
4. n_j^2 slots $\Rightarrow$ **no collisions** (birthday-paradox argument)

Total space:

$$\sum m_j = 1 + 4 + 0 + 1 + 9 = 15$$

Expected: $\mathcal{O}(n)$

Key Idea

Each level-2 function h_j is picked from a universal family and re-sampled until it is **collision-free**. Result: $\mathcal{O}(1)$ **worst-case search** with $\mathcal{O}(n)$ space.

Part 7

Practice: Python & C++

Python: `dict` and `set`

Python

```python
# dict: key -> value mapping
phone = {"Alice": "0612", "Bob": "0698"}
phone["Carol"] = "0645"            # insert
print(phone["Alice"])              # search: O(1) avg
del phone["Bob"]                   # delete: O(1) avg

# set: unique keys, no values
seen = set()
seen.add(42)
print(42 in seen)                  # True - O(1) avg
seen.discard(42)                   # remove if present
```

- **Implementation:** open addressing with perturbation (not pure linear probing)
- Load factor kept $\leqslant 2/3$; resizes by $\times 4$ (small) or $\times 2$ (large)
- Hash function: SipHash (keyed, randomized per process via `PYTHONHASHSEED`)

C++: unordered_map and unordered_set

Cpp

```cpp
#include <unordered_map>
#include <unordered_set>
#include <string>

std::unordered_map<std::string, int> ages;
ages["Alice"] = 22;                  // insert
ages.emplace("Bob", 25);             // insert (no overwrite)
auto it = ages.find("Alice");        // search: O(1) avg
if (it != ages.end())
    std::cout << it->second;         // 22
ages.erase("Bob");                   // delete

std::unordered_set<int> seen;
seen.insert(42);
seen.count(42);  // 1 if present, 0 otherwise
```

- **Implementation:** chaining (linked lists per bucket)
- Default max load factor: 1.0; rehash when exceeded

Custom Hash Functions in C++

Cpp

```cpp
struct Point { int x, y; };

// Custom hash for Point
struct PointHash {
    size_t operator()(const Point& p) const {
        // Combine hashes (boost::hash_combine pattern)
        size_t h = std::hash<int>{}(p.x);
        h ^= std::hash<int>{}(p.y)
             + 0x9e3779b9 + (h << 6) + (h >> 2);
        return h;
    }
};
struct PointEq {
    bool operator()(const Point& a,
                    const Point& b) const {
        return a.x == b.x && a.y == b.y;
    }
};
std::unordered_set<Point, PointHash, PointEq> pts;
```

0x9e3779b9 $\approx 2^{32}/\varphi$ (golden ratio) — good bit mixing.

Hash Tables vs. BSTs — When to Choose What?

Criterion	Hash Table	Balanced BST
Average lookup	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
Worst-case lookup	$\mathcal{O}(n)$*	$\mathcal{O}(\log n)$ guaranteed
Ordered iteration	✗ No	✓ Yes (in-order)
Range queries	✗ No	✓ $\mathcal{O}(\log n + k)$
Memory overhead	Table + (chains)	3 pointers/node
Cache behavior	Good (open addr.)	Poor (pointer chasing)

* $\mathcal{O}(1)$ with universal hashing or perfect hashing for static sets

Key Idea

Rule of thumb: use hash tables for pure lookup; use BSTs when you need **ordering** (next week: BSTs & Red-Black Trees, CLRS 12–13).

Part 8

Summary & Takeaways

Chapter Summary

1. **Direct-address tables** — $\mathcal{O}(1)$ everything, but $\Theta(|U|)$ space
2. **Hash tables with chaining** — $\mathcal{O}(1+\alpha)$ average; works well when $\alpha = \mathcal{O}(1)$
3. **Hash functions** — division, multiplication, **universal hashing** (Carter–Wegman) for guaranteed $\mathcal{O}(1)$ expected
4. **Open addressing** — no extra memory, but $\alpha < 1$ required; $\leqslant 1/(1-\alpha)$ probes for unsuccessful search
5. **Perfect hashing** — $\mathcal{O}(1)$ worst-case for static sets, $\mathcal{O}(n)$ space via two-level scheme

Key Idea

Hash tables are the most widely used data structure after arrays. Understanding load factor, collision resolution, and universal hashing is essential for every programmer.

Key Complexity Results

Operation / Setting	Average	Worst
Direct addressing (all ops)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Chaining — unsuccessful search	$\Theta(1+\alpha)$	$\Theta(n)$
Chaining — successful search	$\Theta(1+\alpha)$	$\Theta(n)$
Chaining — insert	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Chaining — delete (doubly linked)	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Open addr. — unsuccessful search	$\leqslant 1/(1-\alpha)$	$\Theta(n)$
Open addr. — successful search	$\leqslant \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$	$\Theta(n)$
Perfect hashing — search	$\mathcal{O}(1)$	$\mathcal{O}(1)$

CLRS Exercises to Practice

Recommended exercises from CLRS Chapter 11:

Fundamentals:

- 11.1-1: Max in a direct-address table
- 11.2-2: Insert with chaining ($h(k) = k \bmod 9$)
- 11.2-3: Sorted chains — does it help?
- 11.3-4: Multiplication method example

Deeper understanding:

- 11.2-1: Expected number of collisions
- 11.4-1: Linear probing & double hashing
- 11.4-3: Bounds at $\alpha = 3/4$ and $7/8$
- 11.2-5: Worst-case $\Theta(n)$ search

Next week: Binary Search Trees & Red-Black Trees (CLRS 12–13) — when you need *ordering*, not just lookup.

References

- **CLRS** (4th ed.), Chapter 11 — Hash Tables
- **CLRS** (4th ed.), Chapter 19 — Data Structures for Disjoint Sets (Union-Find, used with hashing in some applications)
- Carter & Wegman (1979), "Universal classes of hash functions"
- Fredman, Komlós, Szemerédi (1984), "Storing a sparse table with $O(1)$ worst-case access time"
- **Python docs:** `https://docs.python.org/3/library/stdtypes.html#mapping-types-dict`
- **cppreference:** `https://en.cppreference.com/w/cpp/container/unordered_map`