PSL UNIVERSITÉ PARIS

Hash Tables

The Art of Computer Programming 2 — Week 3 — March 4, 2026

Objectives

By the end of this lab, you should be able to:

- Trace hash-table operations by hand (insertion, search, deletion)
- Implement a hash table with chaining from scratch in Python
- Analyse the impact of the load factor α on performance
- Compare hash functions (division, multiplication, universal)
- Implement open addressing with linear probing and double hashing
- Empirically verify the theoretical average-case bounds from CLRS

1 Warm-up: Hashing by Hand (15 min)

Pen and paper — no computer needed.

1.1 Chaining (CLRS Exercise 11.2-2, adapted)

Consider a hash table with $m = 9$ slots and the hash function $h(k) = k \bmod 9$. Insert the following keys **in order**, resolving collisions by chaining (prepend to the list):

$$5,\ 28,\ 19,\ 15,\ 20,\ 33,\ 12,\ 17,\ 10$$

(a) Compute $h(k)$ for each key and fill in the table below.

(b) Draw the final state of the hash table (all 9 slots, with their chains).

(c) What is the load factor α?

(d) What is the length of the longest chain?

Key k	5	28	19	15	20	33	12	17	10
$h(k) = k \bmod 9$									

1.2 Expected number of collisions

We hash n distinct keys into a table of size m using independent uniform hashing.

(a) How many pairs of distinct keys are there?

(b) What is the probability that a given pair (k_i, k_j) collides (i.e., $h(k_i) = h(k_j)$)?

(c) Using linearity of expectation, derive the expected number of collisions: $\binom{n}{2}/m$.

(d) Compute the expected collisions for $n = 9, m = 9$ and for $n = 20, m = 9$.

Hint

This is CLRS Exercise 11.2-1. Define an indicator random variable X_{ij} for each pair.

1.3 Sorted chains (CLRS 11.2-3)

Professor Marley proposes keeping each chain in **sorted order**. How does this affect:

(a) Successful search time?

(b) Unsuccessful search time?

(c) Insertion time?

(d) Deletion time?

2 Implement a Hash Table with Chaining 💻 (20 min)

2.1 Basic implementation

Complete the following hash table implementation:

```python
class ChainingHashTable:
    """Hash table with collision resolution by chaining."""

    def __init__(self, m=8):
        self.m = m
        self.table = [[] for _ in range(m)]  # list of lists
        self.n = 0   # number of stored elements

    def _hash(self, key):
        return hash(key) % self.m

    def insert(self, key, value):
        """Insert (key, value). Update value if key already exists."""
        # YOUR CODE HERE
        pass

    def search(self, key):
        """Return the value associated with key, or None."""
        # YOUR CODE HERE
        pass

    def delete(self, key):
        """Delete the entry with the given key. Return True if found."""
        # YOUR CODE HERE
        pass

    @property
    def load_factor(self):
        return self.n / self.m

    def display(self):
        print(f"Hash Table (m={self.m}, n={self.n}, "
              f"alpha={self.load_factor:.2f})")
        for i, chain in enumerate(self.table):
            entries = ' -> '.join(f'({k}: {v})' for k, v in chain)
            print(f"  [{i}] {entries if entries else 'empty'}")
```

2.2 Test your implementation

```python
ht = ChainingHashTable(m=7)

# Insert
for name, grade in [("Alice", 18), ("Bob", 14), ("Carol", 16),
                    ("David", 12), ("Eve", 19), ("Frank", 15)]:
    ht.insert(name, grade)

# Search
assert ht.search("Alice") == 18
assert ht.search("Eve") == 19
assert ht.search("Zoe") is None

# Update
ht.insert("Alice", 20)
assert ht.search("Alice") == 20

# Delete
assert ht.delete("Bob") == True
assert ht.search("Bob") is None
assert ht.delete("Bob") == False  # already deleted

ht.display()
print("All tests passed!")
```

2.3 Dynamic resizing

Extend your hash table with **automatic resizing**: when the load factor exceeds a threshold (e.g., $\alpha > 0.75$), double the table size and rehash all elements.

```python
def _resize(self, new_m):
    """Resize the table to new_m slots and rehash all elements."""
    # YOUR CODE HERE
    pass
```

Hint

Save the old table, create a new empty table of size `new_m`, then re-insert every element. Don't forget to reset `self.n` before re-inserting!

Questions:

1. What is the amortized cost of insertion with resizing?
2. Insert $n = 100$ elements into a table starting with $m = 4$. How many times does the table resize? What is its final size?

3 Hash Function Quality (15 min)

Recall

Division method: $h(k) = k \bmod m$. Works best when m is prime.
Multiplication method: $h(k) = \lfloor m \cdot (kA \bmod 1) \rfloor$ with $A \approx (\sqrt{5} - 1)/2$.

3.1 Why m matters for the division method

(a) Create a list of 300 keys that are **multiples of 6**: `keys = [6*i for i in range(300)]`.

(b) Hash them with $h(k) = k \bmod m$ for:

- $m = 30$ (composite, divisible by 6)
- $m = 31$ (prime)
- $m = 32$ (power of 2)

(c) For each choice of m, count how many elements land in each slot. Plot a bar chart.

(d) Which m gives the best distribution? The worst? Explain.

Hint

Use `matplotlib` to visualize. Compute and display the max chain length and number of empty slots for each m.

3.2 Multiplication method

Repeat the experiment above using the multiplication method with $A = (\sqrt{5} - 1)/2$ and $m = 32$. Compare the distribution with the division method using $m = 32$.

3.3 Adversarial keys

For **any** fixed hash function $h(k) = k \bmod m$, construct a set of $n = 100$ keys that **all hash to slot 0**. What does this imply about the worst-case search time?

4 Universal Hashing (15 min)

Recall

A family $\mathcal{H}$ of hash functions is **universal** if for all distinct $k_1, k_2 \in U$:

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

The Carter–Wegman family: $h_{a,b}(k) = ((ak+b) \bmod p) \bmod m$ with p prime $\geqslant |U|$, $a \in \{1, \ldots, p-1\}$, $b \in \{0, \ldots, p-1\}$.

4.1 Implement the Carter–Wegman family

```python
import random

class UniversalHash:
    """Carter-Wegman universal hash function."""
    def __init__(self, m, p=104_729):
        self.m = m
        self.p = p
        self.a = random.randint(1, p - 1)
        self.b = random.randint(0, p - 1)

    def __call__(self, k):
        return ((self.a * k + self.b) % self.p) % self.m
```

4.2 Verify the universality property

Experimentally verify that $\Pr[h(k_1) = h(k_2)] \leqslant 1/m$ for two distinct keys k_1, k_2:

1. Fix $k_1 = 42$ and $k_2 = 137$ and $m = 31$.
2. Sample 50 000 random (a, b) pairs and count how often $h_{a,b}(k_1) = h_{a,b}(k_2)$.
3. Compare the empirical collision probability with $1/m$.

4.3 Universal hashing vs. adversarial keys

Take the adversarial keys from Section 3.3 (multiples of m) and hash them using a random Carter–Wegman function. Plot the distribution. Is the adversary defeated?

5 Open Addressing (20 min)

Recall

In open addressing, all elements are stored **in the table itself** (no chains). On collision, we probe other slots according to a **probe sequence**:

- **Linear probing:** $h(k, i) = (h'(k) + i) \bmod m$
- **Double hashing:** $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$

The table **must** have $\alpha < 1$ (more slots than elements).

5.1 Open addressing by hand

Consider a table of size $m = 11$ with $h'(k) = k \bmod 11$. Insert the keys in order using **linear probing**:

$$10,\ 22,\ 31,\ 4,\ 15,\ 28,\ 17,\ 88,\ 59$$

(a) For each key, write down the probe sequence and the final slot.

(b) How many probes does each insertion require?

(c) Identify the **clusters**. How many are there, and what are their lengths?

5.2 Implement open addressing

```python
class OpenAddressingHashTable:
    EMPTY = None
    DELETED = "__DELETED__"

    def __init__(self, m=16, probe='linear'):
        self.m = m
        self.table = [self.EMPTY] * m
        self.n = 0
        self.probe = probe

    def _h1(self, key):
        return hash(key) % self.m

    def _h2(self, key):
        """For double hashing: must be coprime with m."""
        return 1 + (hash(key) % (self.m - 1))

    def _probe(self, key, i):
        if self.probe == 'linear':
            return (self._h1(key) + i) % self.m
        elif self.probe == 'double':
            return (self._h1(key) + i * self._h2(key)) % self.m

    def insert(self, key):
        """Insert key. Return (slot, num_probes)."""
        # YOUR CODE HERE
        pass

    def search(self, key):
        """Search for key. Return (slot_or_None, num_probes)."""
        # YOUR CODE HERE
        pass
```

Important

When searching, stop when you hit an `EMPTY` slot (key is absent), but **skip over** `DELETED` slots (a key might be beyond them). This is why deletion in open addressing uses tombstones.

5.3 Linear probing vs. double hashing

1. Insert 25 random keys (from `range(1000)`) into a table of size $m = 32$ using linear probing and double hashing.
2. Print the average number of probes for each method.
3. Visualize both tables (e.g., a horizontal bar showing occupied/empty slots). Which has more clustering?

5.4 Probes vs. load factor

Measure the average number of probes per insertion as a function of α:

1. Use $m = 1009$ (prime). For each $\alpha \in \{0.1, 0.2, \ldots, 0.9, 0.95\}$, insert $n = \lfloor \alpha \cdot m \rfloor$ random keys.
2. Record the average number of probes for linear probing and double hashing.
3. On the same plot, draw the theoretical curves:

- Unsuccessful search: $\frac{1}{1-\alpha}$ (Theorem 11.6)
- Successful search: $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ (Theorem 11.8)

4. At what load factor does performance start to degrade significantly?

6 Python `dict` Internals (5 min)

6.1 Explore Python's `hash()`

```python
import sys

# Integer hashing
for n in [0, 1, 42, -1, -2, 2**61 - 1]:
    print(f"hash({n:>20}) = {hash(n)}")

# String hashing (randomized since Python 3.3)
print(f"\nhash('hello') = {hash('hello')}")
print(f"hash_info = {sys.hash_info}")
```

Questions:

1. For small integers, what is `hash(n)`?
2. Why is `hash(-1) == -2`?
3. Why does `hash("hello")` change between executions?
4. What types are hashable? What types are not? Why?

6.2 Observe automatic resizing

```python
d = {}
for i in range(100):
    d[i] = i
    if i > 0 and sys.getsizeof(d) != sys.getsizeof({j: j for j in range(i)}):
        # This is a rough way; just print sizes
        pass
    if i < 5 or i % 10 == 0:
        print(f"n={i+1:>3}, size={sys.getsizeof(d)} bytes")
```

At what values of n does the dictionary resize? What is the approximate growth factor?

Bonus Exercises

For students who finish early.

Bonus 1 — Bloom Filter

Implement a Bloom filter with k hash functions using double hashing:

```python
class BloomFilter:
    def __init__(self, size, num_hashes):
        self.bits = [False] * size
        self.size = size
        self.k = num_hashes

    def add(self, item):
        # YOUR CODE: set k bits to True
        pass

    def might_contain(self, item):
        # YOUR CODE: return False if any bit is 0
        pass
```

1. Insert 10 000 random strings. Test 10 000 *different* strings and measure the false-positive rate.
2. Compare with the theoretical rate $(1 - e^{-kn/m})^k$.
3. Plot the false-positive rate as a function of k for fixed n and m.

Bonus 2 — HashDoS Simulation

Simulate a HashDoS attack:

1. Write a naïve deterministic hash: `h(s) = (31*h + ord(c)) for c in s`.
2. Find 500 strings that all hash to the same slot (brute-force search).
3. Compare the insertion time of 500 random strings vs. 500 adversarial strings.
4. Explain why Python $\geqslant 3.3$ uses SipHash with a random seed.

Bonus 3 — Cuckoo Hashing ⋆

Implement cuckoo hashing with two tables T_1, T_2 and two hash functions h_1, h_2:

- INSERT(x): place x in $T_1[h_1(x)]$; if occupied, evict the occupant to its *other* table, and repeat.
- SEARCH(k): check $T_1[h_1(k)]$ and $T_2[h_2(k)]$ — $\mathcal{O}(1)$ worst case!
- If an insertion cycle is detected (after $\mathcal{O}(\log n)$ evictions), rehash with new h_1, h_2.

Test with $n = 1000$ keys and tables of size $m = 1500$ each.

Key Formulas

Load factor	$\alpha = n/m$
Chaining — unsuccessful search	$\Theta(1+\alpha)$ expected
Chaining — successful search	$\Theta(1+\alpha)$ expected
Open addr. — unsuccessful	$\leqslant \dfrac{1}{1-\alpha}$ expected probes
Open addr. — successful	$\leqslant \dfrac{1}{\alpha} \ln \dfrac{1}{1-\alpha}$ expected probes
Universal hashing	$\Pr[h(k_1) = h(k_2)] \leqslant 1/m$
Bloom filter FP rate	$\approx (1 - e^{-kn/m})^k$

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