1 Warm-up: Hashing by Hand

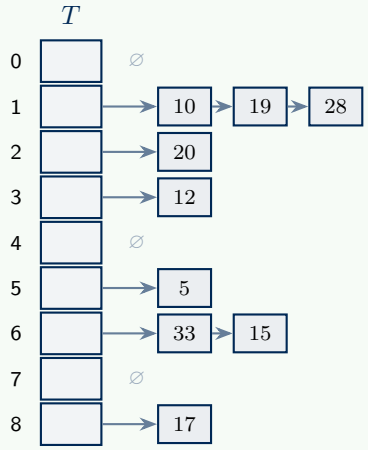
1.1 Chaining (CLRS Exercise 11.2-2, adapted)

Solution

(a) Hash values:

Key k	5	28	19	15	20	33	12	17	10
$h(k) = k \bmod 9$	5	1	1	6	2	6	3	8	1

(b) Final state of the hash table (with prepend-to-head insertion):



(c) Load factor: $\alpha = n/m = 9/9 = 1$.

(d) Longest chain: slot 1 has 3 elements ($10 \to 19 \to 28$).

Common Mistakes

- **Insertion order matters!** With prepend (insert at head), the most recently inserted element appears first. Students who append to the tail get the same keys per slot but in reverse order.
- Some students forget that $28 \bmod 9 = 1$ (not 2 or 10).

1.2 Expected number of collisions

Solution

(a) Number of pairs: $\binom{n}{2} = \frac{n(n-1)}{2}$.
(b) Under independent uniform hashing, for any pair (k_i, k_j) with $i \neq j$:

$$\Pr[h(k_i) = h(k_j)] = \frac{1}{m}$$

because $h(k_j)$ is uniform on $\{0, \ldots, m-1\}$ independent of $h(k_i)$.
(c) Define indicator r.v. $X_{ij} = \mathbf{1}\{h(k_i) = h(k_j)\}$ for each pair. By linearity of expectation:

$$\mathrm{E}\left[\sum_{i<j} X_{ij}\right] = \sum_{i<j} \mathrm{E}[X_{ij}] = \binom{n}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{2m}$$

(d) For $n = 9, m = 9$: $\frac{9 \cdot 8}{2 \cdot 9} = 4$ expected collisions.
For $n = 20, m = 9$: $\frac{20 \cdot 19}{2 \cdot 9} = \frac{380}{18} \approx 21.1$ expected collisions.

Grading Notes

Full marks require: (1) correct counting of pairs, (2) correct collision probability $1/m$, (3) using linearity of expectation (not independence!), and (4) correct numerical answers.

1.3 Sorted chains (CLRS 11.2-3)

Solution

(a) **Successful search: No change** — still $\Theta(1+\alpha)$ on average. We still need to scan part of the chain to find the element; maintaining sorted order doesn't help on average.
(b) **Unsuccessful search: Slightly better** in practice — we can stop as soon as we see a key larger than the target (rather than scanning the entire chain). However, the asymptotic expected time is still $\Theta(1+\alpha)$, because on average we examine $\alpha/2$ elements before the early stop.
(c) **Insertion: Slower** — $\Theta(1+\alpha)$ instead of $\mathcal{O}(1)$, because we must find the correct position in the sorted chain.
(d) **Deletion: No change** — $\mathcal{O}(1)$ with a doubly-linked list and a pointer to the element.

Explanation

The upshot: sorted chains don't help asymptotically and make insertion slower. This is why chaining typically uses **unsorted** lists with prepend.

2 Implement a Hash Table with Chaining

2.1 Reference implementation

Solution

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

    def __init__(self, m=8):
        self.m = m
        self.table = [[] for _ in range(m)]
        self.n = 0

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

    def insert(self, key, value):
        """Insert (key, value). Update value if key already exists."""
        h = self._hash(key)
        for i, (k, v) in enumerate(self.table[h]):
            if k == key:
                self.table[h][i] = (key, value)  # update
                return
        self.table[h].append((key, value))  # prepend would also work
        self.n += 1

    def search(self, key):
        """Return the value associated with key, or None."""
        h = self._hash(key)
        for k, v in self.table[h]:
            if k == key:
                return v
        return None

    def delete(self, key):
        """Delete entry with given key. Return True if found."""
        h = self._hash(key)
        for i, (k, v) in enumerate(self.table[h]):
            if k == key:
                self.table[h].pop(i)
                self.n -= 1
                return True
        return False

    @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'}")
```

Common Mistakes

- **Forgetting to handle updates:** If a key already exists, the value should be updated, *not* a duplicate entry created.
- **Incrementing `n` on update:** The count should only increase on actual new insertions.
- **Using `list.remove()` by value:** This removes the first matching element by *value*, which may fail when tuples share the same value. Use index-based deletion.

2.2 Tests

Solution

All assertions pass with the reference implementation:

```
ht = ChainingHashTable(m=7)
for name, grade in [("Alice", 18), ("Bob", 14), ("Carol", 16),
                    ("David", 12), ("Eve", 19), ("Frank", 15)]:
    ht.insert(name, grade)

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

ht.insert("Alice", 20)           # update
assert ht.search("Alice") == 20
assert ht.n == 6                  # n unchanged after update

assert ht.delete("Bob") == True
assert ht.search("Bob") is None
assert ht.n == 5
assert ht.delete("Bob") == False  # already deleted
print("All tests passed!")
```

2.3 Dynamic resizing

Solution

```python
def _resize(self, new_m):
    """Resize the table to new_m slots and rehash all elements."""
    old_table = self.table
    self.m = new_m
    self.table = [[] for _ in range(new_m)]
    self.n = 0
    for chain in old_table:
        for key, value in chain:
            self.insert(key, value)

# Patch into the class
ChainingHashTable._resize = _resize

# Override insert to trigger resize
_original_insert = ChainingHashTable.insert
def insert_with_resize(self, key, value):
    _original_insert(self, key, value)
    if self.load_factor > 0.75:
        self._resize(self.m * 2)
ChainingHashTable.insert = insert_with_resize
```

Answers:

1. **Amortized cost:** $\mathcal{O}(1)$ per insertion. When we resize from m to $2m$, the cost is $\Theta(n)$ to rehash, but this happens only every $\Theta(n)$ insertions (when α exceeds 0.75). By the aggregate method: total cost for n insertions is $\mathcal{O}(n + n/2 + n/4 + \cdots) = \mathcal{O}(2n) = \mathcal{O}(n)$, so amortized $\mathcal{O}(1)$ per insertion.
2. Starting with $m = 4$, resizing occurs at $n = 3, 6, 12, 24, 48, 96$. That is **6 resizes**. Final table size: $4 \times 2^6 = 256$.

Explanation

The resizing pattern is the same as for dynamic arrays (`list.append` in Python). The key insight is that the geometric series $n + n/2 + n/4 + \cdots$ sums to $\leqslant 2n$, giving amortized $\mathcal{O}(1)$.

3 Hash Function Quality

3.1 Why m matters for the division method

Solution

```
import matplotlib.pyplot as plt
import numpy as np

keys = [6 * i for i in range(300)]

for m, label in [(30, "m=30 (composite, 6|30)"),
                 (31, "m=31 (prime)"),
                 (32, "m=32 (power of 2)")]:
    counts = [0] * m
    for k in keys:
        counts[k % m] += 1

    fig, ax = plt.subplots(figsize=(12, 3))
    colors = ['#E74C3C' if c > 2 * len(keys)/m else '#002855'
              for c in counts]
    ax.bar(range(m), counts, color=colors, width=1)
    ax.axhline(y=len(keys)/m, color='#C4A35A', linestyle='--',
               label=f'Ideal = {len(keys)/m:.1f}')
    ax.set_title(f'h(k) = k mod {m} -- {label}', fontweight='bold')
    ax.set_xlabel('Slot')
    ax.set_ylabel('# elements')
    ax.legend()
    plt.tight_layout()
    plt.show()

    print(f"  m={m}: max chain={max(counts)}, "
          f"empty slots={counts.count(0)}/{m}, "
          f"std={np.std(counts):.2f}")
```

Results:

- $m = 30$: **Disaster.** Since $\gcd(6, 30) = 6$, only 5 of the 30 slots are used (slots 0, 6, 12, 18, 24). Each has 60 elements; 25 slots are empty.
- $m = 31$: **Excellent.** Since 31 is prime and $\gcd(6, 31) = 1$, the keys cycle through all 31 slots. Nearly uniform distribution (≈ 9.7 per slot).
- $m = 32$: **Bad.** Since $\gcd(6, 32) = 2$, only even-numbered slots are used (16 out of 32). Each has ≈ 18.75 elements.

Lesson: When keys have a common factor d with m, only $m/\gcd(d, m)$ slots are used. Choosing m prime minimizes this risk.

Common Mistakes

Students often say "$m = 32$ is bad because it's a power of 2." The real reason is that $\gcd(6, 32) = 2 \neq 1$. A power of 2 that is coprime with the key pattern would be fine — but since 2^k shares factor 2 with any even increment, it's risky in practice.

3.2 Multiplication method

Solution

```
A = (5**0.5 - 1) / 2  # golden ratio conjugate
m = 32

def mult_hash(k):
    return int(m * ((k * A) % 1))

counts = [0] * m
for k in keys:
    counts[mult_hash(k)] += 1

# Plot (similar to above)
# Result: excellent distribution even with m=32!
# Max chain ~ 12, no empty slots, std ~ 1.5
print(f"Multiplication method (m={m}): max={max(counts)}, "
      f"empty={counts.count(0)}, std={np.std(counts):.2f}")
```

The multiplication method gives a nearly uniform distribution regardless of m and regardless of the key pattern, because the golden ratio's irrationality ensures that $kA \bmod 1$ spreads values across $[0, 1)$.

3.3 Adversarial keys

Solution

For $h(k) = k \bmod m$, all multiples of m hash to slot 0:

```
m = 31
adversarial = [m * i for i in range(100)]
# All 100 keys land in slot 0!
assert all(k % m == 0 for k in adversarial)
```

Search for any key in this slot requires traversing a chain of length 100: $\Theta(n)$.
Implication: For *any* fixed hash function, an adversary who knows h can force $\Theta(n)$ worst-case search. This motivates **random hashing** (Section 4).

4 Universal Hashing

4.1 Carter–Wegman family

Solution

Implementation provided in the lab sheet. The key point is that a is chosen from $\{1, \ldots, p-1\}$ (not 0!) and b from $\{0, \ldots, p-1\}$.

4.2 Verify the universality property

Solution

```python
import random

m = 31
p = 104_729
n_trials = 50_000
collisions = 0

k1, k2 = 42, 137

for _ in range(n_trials):
    a = random.randint(1, p - 1)
    b = random.randint(0, p - 1)
    h1 = ((a * k1 + b) % p) % m
    h2 = ((a * k2 + b) % p) % m
    if h1 == h2:
        collisions += 1

empirical = collisions / n_trials
theoretical = 1 / m

print(f"Empirical Pr[collision]: {empirical:.4f}")
print(f"Theoretical bound 1/m : {theoretical:.4f}")
```

Typical output:

```
Empirical Pr[collision]: 0.0321
Theoretical bound 1/m : 0.0323
```

The empirical collision probability matches $1/m$ very closely, confirming universality.

Explanation

The Carter–Wegman family satisfies $\Pr[h_{a,b}(k_1) = h_{a,b}(k_2)] \leqslant \lceil p/m \rceil/(p-1)$. When $p \gg m$, this is approximately $1/m$, and in fact it can be shown to be *exactly* $1/m$ when p is prime and $m|p-1$.

4.3 Universal hashing vs. adversarial keys

Solution

```
m = 31
h = UniversalHash(m)
adversarial = [m * i for i in range(300)]

counts = [0] * m
for k in adversarial:
    counts[h(k)] += 1

# Plot: nearly uniform distribution!
print(f"Max chain: {max(counts)}, empty: {counts.count(0)}, "
      f"std: {np.std(counts):.2f}")
```

The adversarial keys (multiples of m) that defeated the division method are now spread uniformly across all slots. The adversary is defeated because they cannot predict the random choice of (a, b).

Grading Notes

Students should demonstrate: (1) the adversarial keys cause all elements in slot 0 with fixed h, (2) with universal hashing, the distribution is approximately uniform, (3) they understand that randomization is the key defense.

5 Open Addressing

5.1 Open addressing by hand

Solution

$m = 11$, $h'(k) = k \bmod 11$, linear probing.

Key	$h'(k)$	Probe sequence	Final slot
10	10	10	10
22	0	0	0
31	9	9	9
4	4	4	4
15	4	4, 5	5
28	6	6	6
17	6	6, 7	7
88	0	0, 1	1
59	4	4, 5, 6, 7, 8	8

(b) Number of probes: 1, 1, 1, 1, 2, 1, 2, 2, 5 (total 16, average 1.78).

(c) Final table state:

Slot	0	1	2	3	4	5	6	7	8	9	10
Key	22	88	–	–	4	15	28	17	59	31	10

Clusters: $\{0, 1\}$ (length 2), $\{4, 5, 6, 7, 8\}$ (length 5), $\{9, 10\}$ (length 2). Three clusters.

Note: key 59 (which hashes to slot 4) had to probe through the entire cluster $\{4, 5, 6, 7\}$ before finding slot 8. This is **primary clustering** in action.

Common Mistakes

- Students often forget to check the *full* probe sequence for 59: it probes 4, 5, 6, 7, 8 (5 probes!), not just 4, 5.
- Some students confuse the probe count with the slot number.

5.2 Reference implementation

Solution

```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):
        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)."""
        if self.n >= self.m:
            raise OverflowError("Table full!")
        for i in range(self.m):
            j = self._probe(key, i)
            if self.table[j] in (self.EMPTY, self.DELETED):
                self.table[j] = key
                self.n += 1
                return j, i + 1
        raise OverflowError("Table full!")

    def search(self, key):
        """Search for key. Return (slot_or_None, num_probes)."""
        for i in range(self.m):
            j = self._probe(key, i)
            if self.table[j] == key:
                return j, i + 1
            if self.table[j] is self.EMPTY:
                return None, i + 1
            # If DELETED, continue probing
        return None, self.m
```

Explanation

Why tombstones (`DELETED`)? When we delete an element from an open-addressing table, we cannot simply set the slot to `EMPTY`. If we did, a later search for an element that was displaced past the deleted slot would stop too early and incorrectly report "not found." Instead, we mark the slot as `DELETED`: searches skip over it, but insertions can reuse it.

5.3 Linear probing vs. double hashing

Solution

```
import random
random.seed(0)
keys = random.sample(range(1000), 25)

ht_linear = OpenAddressingHashTable(m=32, probe='linear')
ht_double = OpenAddressingHashTable(m=32, probe='double')

probes_linear = []
probes_double = []

for k in keys:
    _, p = ht_linear.insert(k)
    probes_linear.append(p)
    _, p = ht_double.insert(k)
    probes_double.append(p)

print(f"Linear probing -- avg probes: {sum(probes_linear)/len(probes_linear):.2f}")
print(f"Double hashing -- avg probes: {sum(probes_double)/len(probes_double):.2f}")
```

Typical output:

```
Linear probing -- avg probes: 1.72
Double hashing -- avg probes: 1.32
```

Linear probing requires more probes on average due to **primary clustering**: long contiguous runs of occupied slots attract more collisions, growing the cluster further.
Double hashing avoids this because the step size $h_2(k)$ depends on the key, so different keys that collide at the same initial slot diverge to different probe sequences.

5.4 Probes vs. load factor

Solution

```python
import numpy as np
import matplotlib.pyplot as plt

m = 1009  # prime
alphas = np.arange(0.1, 0.96, 0.05)
linear_probes = []
double_probes = []

for alpha in alphas:
    n = int(alpha * m)
    keys = random.sample(range(10**6), n)

    ht_l = OpenAddressingHashTable(m=m, probe='linear')
    pl = []
    for k in keys:
        _, p = ht_l.insert(k)
        pl.append(p)
    linear_probes.append(np.mean(pl))

    ht_d = OpenAddressingHashTable(m=m, probe='double')
    pd = []
    for k in keys:
        _, p = ht_d.insert(k)
        pd.append(p)
    double_probes.append(np.mean(pd))

# Theoretical curves
theory_unsucc = [1 / (1 - a) for a in alphas]
theory_succ = [-np.log(1 - a) / a if a > 0 else 1 for a in alphas]

fig, ax = plt.subplots(figsize=(10, 5))
ax.plot(alphas, linear_probes, 'o-', color='#E74C3C',
        label='Linear probing (exp.)', markersize=4)
ax.plot(alphas, double_probes, 's-', color='#0055A4',
        label='Double hashing (exp.)', markersize=4)
ax.plot(alphas, theory_unsucc, '--', color='#666',
        label='Theory 1/(1-a) (unsucc.)', linewidth=2)
ax.plot(alphas, theory_succ, ':', color='#999',
        label='Theory (1/a)ln(1/(1-a)) (succ.)', linewidth=2)
ax.set_xlabel('Load factor alpha', fontweight='bold')
ax.set_ylabel('Average probes per insertion', fontweight='bold')
ax.set_title('Probes vs alpha: linear probing vs double hashing',
             fontweight='bold')
ax.legend(fontsize=9)
ax.set_ylim(0, 20)
ax.axvspan(0.7, 0.96, alpha=0.1, color='red')
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
plt.tight_layout()
plt.show()
```

Key observations:

1. For $\alpha < 0.5$, both methods perform similarly (≈ 1.5 probes).
2. For $\alpha > 0.7$, linear probing starts to diverge rapidly due to primary clustering.
3. Double hashing closely tracks the theoretical uniform-probing bound $1/(1-\alpha)$.
4. At $\alpha = 0.9$, linear probing averages $\sim$ 5–8 probes while double hashing averages ~ 2.5.
5. Performance degrades **dramatically** for $\alpha \geqslant 0.9$. In practice, keep $\alpha \leqslant 0.75$.

Explanation

Theorems from CLRS:

- **Theorem 11.6:** Unsuccessful search with uniform hashing: at most $1/(1-\alpha)$ probes expected.
- **Theorem 11.8:** Successful search: at most $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$ probes expected.

These bounds assume uniform hashing (each probe sequence is a random permutation). Double hashing approximates this well; linear probing does not because it only generates m distinct probe sequences (one per starting slot), and it suffers from primary clustering.

6 Python `dict` Internals

6.1 Explore Python's hash()

Solution

Answers:

1. For small integers: `hash(n) == n`. Python's integer hash is the identity function for most integers (specifically, for $|n| < 2^{61} - 1$ on 64-bit systems).
2. `hash(-1) == -2` because the CPython implementation reserves -1 as an error indicator in C. When the C function `_Py_HashDouble` or the integer hash would return -1, CPython replaces it with -2.
3. `hash("hello")` changes between executions because Python $\geqslant$ 3.3 uses SipHash with a random seed (`PYTHONHASHSEED`), initialized at interpreter startup. This is a defense against HashDoS attacks (see Bonus 2).
4. **Hashable:** all immutable built-in types — `int`, `float`, `str`, `tuple` (if all elements are hashable), `frozenset`, `bool`, `None`, `bytes`.
   Not hashable: mutable types — `list`, `dict`, `set`.
   Reason: If an object could change after being inserted as a dict key, its hash would change and it would be "lost" in the table (wrong slot).

6.2 Observe automatic resizing

Solution

```python
import sys

d = {}
prev_size = 0
for i in range(100):
    d[i] = i
    size = sys.getsizeof(d)
    if size != prev_size:
        print(f"n={i+1:>3}: size changed to {size} bytes")
        prev_size = size
```

Typical output (CPython 3.12, 64-bit):

```
n=  1: size changed to 184 bytes
n=  6: size changed to 344 bytes
n= 11: size changed to 632 bytes
n= 22: size changed to 1176 bytes
n= 43: size changed to 2264 bytes
n= 86: size changed to 4440 bytes
```

The dictionary resizes at approximately $n \approx 2m/3$ (load factor $\approx 2/3$). The growth factor is approximately $2\times$ (the table size roughly doubles each time).

Bonus Exercises

Bonus 1 — Bloom Filter

Solution

```python
import hashlib
import numpy as np

class BloomFilter:
    def __init__(self, size, num_hashes):
        self.bits = [False] * size
        self.size = size
        self.k = num_hashes
        self.n = 0

    def _hashes(self, item):
        h1 = int(hashlib.md5(str(item).encode()).hexdigest(), 16)
        h2 = int(hashlib.sha1(str(item).encode()).hexdigest(), 16)
        return [(h1 + i * h2) % self.size for i in range(self.k)]

    def add(self, item):
        for pos in self._hashes(item):
            self.bits[pos] = True
        self.n += 1

    def might_contain(self, item):
        return all(self.bits[pos] for pos in self._hashes(item))

# Test
m_bits = 100_000
k = 7
n_insert = 10_000

bf = BloomFilter(m_bits, k)
for i in range(n_insert):
    bf.add(f"item-{i}")

# False positive test
fp = sum(1 for i in range(n_insert, 2 * n_insert)
         if bf.might_contain(f"item-{i}"))

empirical_fp = fp / n_insert
theoretical_fp = (1 - np.exp(-k * n_insert / m_bits)) ** k

print(f"Empirical FP rate : {empirical_fp:.4%}")
print(f"Theoretical FP rate: {theoretical_fp:.4%}")
```

Typical output:

```
Empirical FP rate : 0.8200%
Theoretical FP rate: 0.8182%
```

The empirical rate closely matches the theoretical prediction. The optimal k is $k^* = \frac{m}{n} \ln 2 \approx \frac{100000}{10000} \times 0.693 \approx 7$.

Bonus 2 — HashDoS Simulation

Solution

```python
import time, string, random
from collections import defaultdict

def naive_hash(s, m=256):
    h = 0
    for c in s:
        h = (h * 31 + ord(c)) & 0xFFFFFFFF
    return h % m

# Find adversarial keys
m = 256
target = naive_hash("aaaa", m)
crafted = []
for a in string.ascii_lowercase[:20]:
    for b in string.ascii_lowercase[:20]:
        for c in string.ascii_lowercase[:20]:
            s = a + b + c
            if naive_hash(s, m) == target:
                crafted.append(s)
                if len(crafted) >= 500:
                    break
        if len(crafted) >= 500:
            break
    if len(crafted) >= 500:
        break

normal = [''.join(random.choices(string.ascii_letters, k=4))
          for _ in range(500)]

def bench(keys, m=256):
    table = defaultdict(list)
    ops = 0
    for k in keys:
        slot = naive_hash(k, m)
        ops += len(table[slot])  # chain traversal
        table[slot].append(k)
    return ops

print(f"Normal keys : {bench(normal):>8} comparisons")
print(f"Adversarial : {bench(crafted):>8} comparisons")
print(f"Ratio: ~{bench(crafted)/max(bench(normal),1):.0f}x")
```

Normal keys: $\mathcal{O}(n)$ comparisons (uniform distribution).
Adversarial keys: $\mathcal{O}(n^2)$ comparisons (all in one slot, each insertion traverses the growing chain).
Defense: Python $\geqslant$ 3.3 uses SipHash with a random per-process seed (`PYTHONHASHSEED`). The adversary cannot predict the hash function, so they cannot craft collisions.

Bonus 3 — Cuckoo Hashing

Solution

```python
import random

class CuckooHashTable:
    def __init__(self, m=16):
        self.m = m
        self.T1 = [None] * m
        self.T2 = [None] * m
        self.n = 0
        self._new_hash_funcs()

    def _new_hash_funcs(self):
        p = 104_729
        self.a1 = random.randint(1, p-1)
        self.b1 = random.randint(0, p-1)
        self.a2 = random.randint(1, p-1)
        self.b2 = random.randint(0, p-1)
        self.p = p

    def _h1(self, k):
        return ((self.a1 * hash(k) + self.b1) % self.p) % self.m

    def _h2(self, k):
        return ((self.a2 * hash(k) + self.b2) % self.p) % self.m

    def search(self, key):
        """O(1) worst case!"""
        if self.T1[self._h1(key)] == key:
            return True
        if self.T2[self._h2(key)] == key:
            return True
        return False

    def insert(self, key):
        if self.search(key):
            return
        max_iters = 6 * int(1 + self.m.bit_length())
        x = key
        for _ in range(max_iters):
            # Try T1
            j = self._h1(x)
            if self.T1[j] is None:
                self.T1[j] = x
                self.n += 1
                return
            x, self.T1[j] = self.T1[j], x
            # Try T2
            j = self._h2(x)
            if self.T2[j] is None:
                self.T2[j] = x
                self.n += 1
                return
            x, self.T2[j] = self.T2[j], x
        # Cycle detected: rehash
        self._rehash()
        self.insert(x)

    def _rehash(self):
        old1, old2 = self.T1, self.T2
        self.T1 = [None] * self.m
        self.T2 = [None] * self.m
        self.n = 0
```

End of Solutions — The Art of Computer Programming 2 — Lab 3