Open Addressing (CLRS 4th ed., §11.4)

CS5800 — Algorithms

1 Setup

1.1 Generalized hash function

- No pointers, no chains: every slot holds a key or NIL. All n elements live in the table $\Rightarrow \alpha = n/m \leq 1$.
- $h : U \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$; second argument is the *probe number* i.
- Probe sequence of key k: $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$.
- **Requirement:** this sequence must be a permutation of $\langle 0, \ldots, m-1 \rangle$ — every slot gets visited eventually.

1.2 Insert / Search

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

12: function HASH-SEARCH(T, k)
13:     i ← 0
14:     repeat
15:         q ← h(k, i)
16:         if T[q] = k then return q
17:         end if
18:         i ← i + 1
19:     until T[q] = NIL or i = m
20:     return NIL
21: end function
```

- Correctness of SEARCH relies on: insertion never leaves a "hole" inside a key's probe sequence that stops search early.

1.3 Delete

- Cannot write NIL into $T[q]$: some other key k' may have probed through q (found it occupied) and landed further along. Zeroing q breaks SEARCH(T, k').
- Fix: sentinel DELETED. INSERT may overwrite DELETED; SEARCH treats DELETEDas "keep probing," not as a stop condition.
- Cost: DELETEDslots inflate probe lengths over time ⇒ chaining preferred under heavy deletion.

1: **function** HASH-DELETE(T, k)
2: $q \leftarrow$ HASH-SEARCH(T, k)
3: **if** $q \neq$ NIL **then** $T[q] \leftarrow$ DELETED
4: **end if**
5: **end function**

---

2 Probing schemes

All built from an auxiliary ordinary hash function $h'(k)$ (or h_1, h_2).

2.1 Linear probing

- $h(k, i) = (h'(k) + i) \bmod m$.
- Only m distinct probe sequences (determined by $h'(k)$ alone).
- **Primary clustering**: runs of occupied slots grow; any key hashing into the run must walk it.

2.2 Quadratic probing

- $h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, $c_1, c_2 \neq 0$.
- Need c_1, c_2, m constrained (e.g. $m = 2^r$) to actually hit all m slots.
- Same $h'(k)$ ⇒ identical later probes ⇒ **secondary clustering**. Still only m distinct sequences.

2.3 Double hashing

- $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$.
- Need $h_2(k)$ coprime to m: take m prime with $1 < h_2(k) < m$, or $m = 2^r$ with $h_2(k)$ odd.
- $\Theta(m^2)$ distinct probe sequences (depends on pair $(h_1(k), h_2(k))$) ⇒ closest of the three to uniform hashing.

2.4 Uniform hashing (analysis idealization)

- Each key's probe sequence equally likely to be any of the $m!$ permutations. No scheme achieves this exactly; double hashing approximates it best.

Theorem 1 (11.6)**.** *Under uniform hashing, $\alpha = n/m < 1$, expected # probes in an unsuccessful search* $\leq \dfrac{1}{1-\alpha}$.

Proof. Pr[probe i occupied | probes $1..i-1$ occupied] $= \dfrac{n-i+1}{m-i+1} \leq \dfrac{n}{m} = \alpha$ (since $n < m$). Expected number of probes is

$$1 + \alpha\big(1 + \alpha(1 + \alpha(\cdots))\big) \leq \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}. \qquad \square$$

Corollary 1 (11.7)**.** *Expected cost of* INSERT $\leq \dfrac{1}{1-\alpha}$ *(insertion = unsuccessful search + one write).*

Theorem 2 (11.8)**.** *Under uniform hashing, expected # probes for a successful search (uniform over the n keys present) is* $\leq \dfrac{1}{\alpha} \ln \dfrac{1}{1-\alpha}$*: average the insertion-time bound* $\frac{1}{1-i/m}$ *over* $i = 0, \ldots, n-1$.

3 Running example

$m = 11$ (prime). $h'(k) = k \bmod 11$. Insert order: $22, 33, 43, 10, 25$.
$h'(22)$=0, $h'(33)$=0, $h'(43)$=10, $h'(10)$=10, $h'(25)$=3.

3.1 Linear probing walk-through

- Insert 22: slot 0 empty $\rightarrow$ placed at 0.
- Insert 33: slot 0 occupied, i=1 $\Rightarrow$ slot 1 empty $\rightarrow$ placed at 1.
- Insert 43: slot 10 empty $\rightarrow$ placed at 10.
- Insert 10: slot 10 occupied (43), i=1 $\Rightarrow$ 0 occ., i=2 $\Rightarrow$ 1 occ., i=3 $\Rightarrow$ 2 empty $\rightarrow$ placed at 2.
- Insert 25: slot 3 empty $\rightarrow$ placed at 3.

Table: slot 0:22, 1:33, 2:10, 3:25, 10:43; rest NIL.

Search 10**:** probe 10 (holds $43 \neq 10$) $\rightarrow$ probe 0 ($22 \neq 10$) $\rightarrow$ probe 1 ($33 \neq 10$) $\rightarrow$ probe 2 ($10 = 10$, found) — exactly retraces the insertion probe sequence, as guaranteed.

Delete 33 **(slot 1), naively:** set $T[1]$ = NIL. Now search(10): probe 10 occ., probe 0 occ., probe 1 = NIL $\Rightarrow$ loop stops, reports "not found" — *wrong*, 10 is still at slot 2. This is exactly the hole-in-the-middle failure motivating DELETED.

Delete 33**, correctly:** $T[1] \leftarrow$ DELETED. Now search(10): probe 10 occ., probe 0 occ., probe 1 = DELETED $\Rightarrow$ keep going (not a stop condition), probe 2: found. Correct.

3.2 Same keys, double hashing

$h_1(k) = k \bmod 11$, $h_2(k) = 1 + (k \bmod 7)$, so $h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod 11$.

- Insert 22: h_1=0, slot 0 empty $\to$ 0.
- Insert 33: h_1=0 occ.; $h_2(33) = 1 + (33 \bmod 7) = 1 + 5 = 6$; i=1 $\Rightarrow (0 + 6) \bmod 11 = 6$, empty $\to$ 6.
- Insert 43: h_1=10, empty $\to$ 10.
- Insert 10: h_1=10 occ.; $h_2(10) = 1 + 3 = 4$; i=1 $\Rightarrow (10 + 4) \bmod 11 = 3$, empty $\to$ 3.
- Insert 25: h_1=3 occ. (by 10 now!); $h_2(25) = 1 + 4 = 5$; i=1 $\Rightarrow (3 + 5) \bmod 11 = 8$, empty $\to$ 8.

Table: 0:22, 3:10, 6:33, 8:25, 10:43 — occupied slots spread out rather than bunched at $\{0, 1, 2, 3\}$ as under linear probing, even though three of the five keys share h_1-collisions with another key. Deletion works identically via DELETED; the correctness argument doesn't depend on which probing scheme produced the sequence.

4 Comparison

	Linear	Quadratic	Double hashing
Formula	$h'(k)+i$	$h'(k)+c_1 i+c_2 i^2$	$h_1(k)+i\,h_2(k)$
Distinct probe sequences	m	m	$\Theta(m^2)$
Clustering	primary	secondary	essentially none
Extra constraints	none	c_1, c_2 vs. m	$\gcd(h_2(k), m) = 1$
Closeness to uniform hashing	worst	middle	best

- All three share the same deletion problem and the same $\alpha \leq 1$ ceiling — these are properties of open addressing itself, not of the probing scheme.
- As $\alpha \to 1$, expected probe counts $\to \infty$ (Thm. 11.6/11.8) for any scheme; open addressing degrades harder near capacity than chaining does.
- Practical takeaway: double hashing dominates linear/quadratic whenever clustering matters and is the usual default; chaining still wins when deletions are frequent, since DELETEDmarkers never get reclaimed under open addressing without a table rebuild.

Advanced ($\star$): Cuckoo Hashing

Not in CLRS §11.4 — Pagh–Rodler 2001. Included because it's the natural next question after double hashing.

Setup

- Two tables T_1, T_2 (size m each), two hash functions h_1, h_2.
- Key k may live *only* at $T_1[h_1(k)]$ or $T_2[h_2(k)]$ — exactly two candidate homes, ever.
- Contrast with double hashing: there, a key has m candidate homes (the whole probe sequence); here, just 2.

Search / Delete

- SEARCH(k): check $T_1[h_1(k)]$, then $T_2[h_2(k)]$. Two probes, always — **worst-case** $O(1)$, not just expected.
- DELETE(k): locate via search, set that slot to NIL directly. *No* DELETED *sentinel needed* — unlike double hashing, no other key's path can be broken, since paths don't exist (only two fixed homes each).

Insert: eviction chain

```
1: function CUCKOO-INSERT(k)
2:     for tries = 1 to MaxLoop do
3:         if T1[h1(k)] = NIL then T1[h1(k)] ← k; return
4:         end if
5:         k ↔ T1[h1(k)]                          ▷ swap k in, evict resident
6:         if T2[h2(k)] = NIL then T2[h2(k)] ← k; return
7:         end if
8:         k ↔ T2[h2(k)]                          ▷ swap, evict resident
9:     end for
10:    rehash with new h1, h2 (or grow tables); reinsert everything
11: end function
```

- Each eviction bounces the displaced key to its *other* home; that may evict someone else, etc. — a chain.
- Chain usually terminates quickly (empty slot found). It can also **cycle** back to k's own home $\Rightarrow$ declare failure, rehash from scratch.
- Cycles become likely once $\alpha \gtrsim 0.5$ (for the 2-table version) — much tighter ceiling than open addressing's $\alpha < 1$.

Running example, continued

T_1, T_2 size 5. $h_1(k) = k \bmod 5$, $h_2(k) = \lfloor k/5 \rfloor \bmod 5$.

- Insert 22: $h_1(22) = 2$, $T_1[2]$ empty $\rightarrow$ placed.
- Insert 33: $h_1(33) = 3$, $T_1[3]$ empty $\rightarrow$ placed.
- Insert 43: $h_1(43) = 3$, occupied by 33. Evict 33: $h_2(33) = \lfloor 33/5 \rfloor \bmod 5 = 6 \bmod 5 = 1$, $T_2[1]$ empty $\rightarrow$ 33 placed there; 43 takes $T_1[3]$.
- Insert 10: $h_1(10) = 0$, $T_1[0]$ empty $\rightarrow$ placed.
- Insert 25: $h_1(25) = 0$, occupied by 10. Evict 10: $h_2(10) = \lfloor 10/5 \rfloor \bmod 5 = 2$, $T_2[2]$ empty $\rightarrow$ 10 placed there; 25 takes $T_1[0]$.

Final: $T_1[0]$=25, $T_1[2]$=22, $T_1[3]$=43; $T_2[1]$=33, $T_2[2]$=10. One eviction chain of length 1 occurred twice; no cycle.

SEARCH(10): $T_1[h_1(10)=0] = 25 \neq 10$, $T_2[h_2(10)=2] = 10$ — found in exactly 2 probes, regardless of table state. Compare to double hashing's 10, which took 2 probes there *by luck of the sequence*, not by guarantee.

Cuckoo vs. double hashing

	Double hashing	Cuckoo hashing
Candidate homes per key	up to m	exactly 2
Search cost	expected $O(1/(1-\alpha))$	**worst-case** $O(1)$ (2 probes)
Insert cost	expected $O(1/(1-\alpha))$	amortized $O(1)$, occasional rehash
Deletion	needs DELETED sentinel	direct NIL, no sentinel
Max useful load α	up to ≈ 1	$\lesssim 0.5$ (2-table version)
Failure mode	degrades gracefully	can cycle $\Rightarrow$ full rehash

- Cuckoo trades table-space efficiency (lower max α) for a hard worst-case search guarantee — valuable when tail latency matters more than average load.
- Both are pointer-free, cache-friendly schemes; cuckoo's two-probe search is friendlier to hardware/parallel lookups (two independent cache lines, checked concurrently).

5 Chaining vs. Open Addressing

Chaining (§11.2)

- Each slot $T[j]$ is a linked list (or other structure) of all keys hashing there.
- + Simple, correct deletion: unlink the node, $O(1)$ given a pointer to it. No sentinel, no degradation.
- + $\alpha = n/m$ can exceed 1 — table doesn't "fill up"; performance degrades smoothly ($\Theta(1+\alpha)$ expected search).
- + Insert is $O(1)$ worst case (prepend to list) — no probing, no failure mode, no rehash trigger.
- + Robust to a bad load factor: cost grows linearly in α, not via a $1/(1-\alpha)$ blowup.
- – Pointer overhead per element (space), and pointer-chasing is cache-unfriendly — list nodes scattered in memory.
- – Needs dynamic memory allocation for nodes; more constant-factor overhead than array slots.

Open addressing (§11.4)

- + No pointers: all data lives in the table array itself. Better cache locality (probes are nearby/computed, not chased).
- + Lower memory overhead per element — just the key (and maybe a status bit), no node/pointer.
- – Hard ceiling $\alpha \leq 1$; performance craters as $\alpha \to 1$ ($1/(1-\alpha) \to \infty$, Thm. 11.6/11.8) — much more load-factor-sensitive than chaining.

- – Deletion is awkward: needs DELETED sentinel (linear/quadratic/double hashing), which *never gets reclaimed* without a full rehash — long-running tables with many deletes silently degrade.
- – Clustering (primary for linear, secondary for quadratic) unless double hashing (or cuckoo) is used.
- Cuckoo hashing (advanced, above) fixes open addressing's search-time variance (worst-case $O(1)$) but pays for it with an even tighter load ceiling ($\alpha \lesssim 0.5$) and a genuine failure mode (cycles $\Rightarrow$ rehash).

When to use which

Use chaining when	Use open addressing when
deletions are frequent	deletions are rare/absent
load factor unpredictable or > 1	n known in advance, α kept moderate
elements are large/variable-size	elements are small, uniform (fit in a slot)
simplicity/correctness prioritized	cache performance / memory footprint matters

- Rule of thumb: open addressing (esp. double hashing or cuckoo) wins in performance-critical, insert/lookup-heavy, low-deletion settings (e.g. compilers' symbol tables during a single pass, static dictionaries). Chaining wins whenever the table's contents churn heavily or α can't be bounded away from 1.