

# ADS2 — Hashing (Week Notes & Full Solutions)

Field	Value
<b>Title</b>	ADS2 — Hashing: Dictionaries, Chaining, Linear Probing, Universal Hashing
<b>Date</b>	2025-11-08
<b>Author</b>	Mads Richardt
<b>Sources used</b>	Weekplan Hashing (weekplan-1.png, weekplan-2.png); Hashing slides by Philip Bille (hashing-01.png...hashing-10.png); Kleinberg–Tardos, Algorithm Design, Ch. 13 §13.6–§13.7
<b>Week plan filename</b>	kt.pdf

## Coverage Table

Weekplan ID	Canonical ID	Title/Label (verbatim)	Assignment Source	Text Source	Status
1.1	—	Insert K into chained hashing ( $m=11$ , $h(k)=k \bmod 11$ )	kt.pdf p.1	slides; KT §13.6	Solved
1.2	—	Insert K into linear probing ( $m=11$ , $h(k)=k \bmod 11$ )	kt.pdf p.1	slides	Solved
1.3	—	Given chained hash table A, can we efficiently find $\max(K)$ ?	kt.pdf p.1	slides; KT §13.6	Solved
1.4	—	Show tables from 1.1 & 1.2 after deleting key 2	kt.pdf p.1	slides	Solved
2	—	Streaming Statistics: distinct IP counter	kt.pdf p.1	slides; KT §13.6	Solved
3	—	Multi-Set Hashing: ADD/REMOVE/REPORT	kt.pdf p.1	slides	Solved
4.1	—	Lazy Deletion in Linear Probing — modify SEARCH/INSERT	kt.pdf p.1	slides	Solved
4.2	—	Lazy vs. eager deletion — pros/cons	kt.pdf p.1	slides	Solved
5	KT §13	BST-sort via random insertion → inorder output	kt.pdf p.1	KT §13	Solved
6.1	—	Dynamic Arrays & Dictionaries when $n \gg m$	kt.pdf p.2	slides; KT §13.6	Solved

Weekplan ID	Canonical ID	Title/Label (verbatim)	Assignment Source	Text Source	Status
6.2	—	Growth-handling with compact space and fast ops	kt.pdf p.2	slides; KT §13.6	Solved
7.1	—	Rabbit Billy: expected bushes until first carrot	kt.pdf p.2	slides	Solved
7.2	—	Rabbit Billy: expected bushes until three carrots	kt.pdf p.2	slides	Solved

## General Methodology and Theory

- **Dictionaries.** Maintain dynamic set  $S \subseteq U$  with SEARCH/INSERT/DELETE.
- **Chained hashing.** Array  $A[0..m-1]$  of lists; store  $x$  at  $A[h(x)]$ . With simple-uniform hashing and load factor  $\alpha = n/m$ , expected time per operation is  $O(1 + \alpha)$ .
- **Linear probing (open addressing).** Keep all keys in  $A$ ; on collision, scan cyclically to the right until an empty slot. Clusters (maximal runs of non-empty cells) drive costs.
- **Universal hashing.** Choose  $h$  at random from a family  $\mathcal{H}$  such that for any distinct  $x, y$ ,  $\Pr[h(x) = h(y)] \leq 1/m$  (e.g.,  $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$  with prime  $p > \max U$ ). Ensures short chains in expectation, independent of input.
- **Resizing.** Keep  $\alpha$  bounded (e.g.,  $\alpha \leq 1$ ) by doubling/halving  $m$  and rehashing. This yields expected **amortized**  $O(1)$  updates and lookups.

## Notes (slides-first)

- **Operations (chaining).** SEARCH: compute  $h(x)$ , scan list  $A[h(x)]$ ; INSERT: add  $x$  to front of list if absent; DELETE: remove  $x$  if present.
- **Operations (linear probing).** SEARCH: start at  $h(x)$  and scan the cluster; INSERT: place at first empty cell at/after  $h(x)$ ; DELETE (eager): remove and re-insert subsequent cluster elements; **Lazy deletion:** place a tombstone that SEARCH treats as occupied and INSERT may reuse.
- **Complexity cheat-sheet.**
- Chaining: time  $O(1 + \alpha)$ ; space  $O(m + n)$ .
- Linear probing: highly cache-efficient; sensitive to  $\alpha$  (keep well below 1).
- **Universal families.** Dot-product modulo prime and affine  $ax + b$  modulo prime are standard universal classes.

## Solutions

### Exercise 1.1 — —

**Assignment Source:** kt.pdf p.1

**Text Source:** slides; KT §13.6

Let  $K = [7, 18, 2, 3, 14, 25, 1, 11, 12, 1332]$  ,  $m = 11$  ,  $h(k) = k \bmod 11$  . Chaining; insert at list front.

Buckets after all insertions (only non-empty shown):

- $A[0] : [11]$
- $A[1] : [1332, 12, 1]$  (since  $1332 \equiv 1$ )
- $A[2] : [2]$
- $A[3] : [25, 14, 3]$
- $A[7] : [18, 7]$

✓ **Answer:** Chained table as listed above; expected chain lengths consistent with slides.

```
Algorithm: chain_insert_front
Input: array A[0..m-1] of lists, key x, hash h
Output: A with x stored

i ← h(x)
if x not in A[i]:
    prepend x to A[i]
// Time: O(1 + |A[i]|)
```

---

### Exercise 1.2 — —

**Assignment Source:** kt.pdf p.1

**Text Source:** slides

Linear probing with the same  $K, m, h$  . Final array (index  $\rightarrow$  value):

- $0 \rightarrow 11, 1 \rightarrow 1, 2 \rightarrow 2, 3 \rightarrow 3, 4 \rightarrow 14, 5 \rightarrow 25, 6 \rightarrow 12, 7 \rightarrow 7, 8 \rightarrow 18, 9 \rightarrow 1332,$   
 $10 \rightarrow \emptyset$  .

✓ **Answer:** As above ( $\emptyset$  denotes empty).

```
Algorithm: lp_insert
Input: array A[0..m-1], key x, hash h
Output: index i where x placed

i ← h(x)
while A[i] ≠ ∅ and A[i] ≠ ⊗: // ⊗ optional tombstone
    i ← (i + 1) mod m
A[i] ← x; return i
// Time: O(cluster length + 1)
```

### Exercise 1.3 — —

**Assignment Source:** kt.pdf p.1

**Text Source:** slides; KT §13.6

**Claim.** From a plain chained table  $A$  (no metadata) one cannot find  $\max(K)$  faster than  $\Theta(n)$  in the worst case; the expected time is also  $\Theta(n)$  under simple-uniform hashing because values are spread across lists.

**Idea.** Any list may contain the maximum; you must inspect all keys. To accelerate, maintain a secondary structure (tracked maximum or a max-heap) updated on INSERT/DELETE.

✓ **Answer:** No. Without satellite data,  $\max(K)$  is  $\Theta(n)$ . With a maintained maximum (or a heap), queries become  $O(1)$  (or  $O(\log n)$ ) with  $O(1)$  update overhead.

---

### Exercise 1.4 — —

**Assignment Source:** kt.pdf p.1

**Text Source:** slides

Delete key 2. - **Chaining (from 1.1):** remove 2 from  $A[2] \rightarrow A[2]$  becomes empty; others unchanged. - **Linear probing (from 1.2, eager deletion):** remove at index 2 and re-insert subsequent cluster items until the first empty cell. Result:  $0 \rightarrow 11, 1 \rightarrow 1, 2 \rightarrow 12, 3 \rightarrow 3, 4 \rightarrow 14, 5 \rightarrow 25, 6 \rightarrow 1332, 7 \rightarrow 7, 8 \rightarrow 18, 9 \rightarrow \emptyset, 10 \rightarrow \emptyset$ .

```
Algorithm: lp_delete_eager
Input: array A[0..m-1], key x, hash h
Output: A with x removed and cluster repaired

find i with A[i]=x; A[i] ← ∅
j ← (i+1) mod m
while A[j] ≠ ∅:
    t ← A[j]; A[j] ← ∅
    lp_insert(A, t, h) // reinsert
    j ← (j+1) mod m
// Time: O(cluster length)
```

✓ **Answer:** Tables exactly as stated above.

---

### Exercise 2 — —

**Assignment Source:** kt.pdf p.1

**Text Source:** slides; KT §13.6

**Goal.** Count the number of **distinct** source IPs in a high-speed stream.

**Solutions. - Exact (baseline):** maintain a hash set of 64-bit IP hashes; space  $O(D)$  for  $D$  distinct IPs; throughput limited by cache. - **Approximate (recommended): HyperLogLog (HLL) / Flajolet–Martin sketch.** Apply  $h : \text{IP} \rightarrow \{0,1\}^{64}$ ; partition by the first  $p$  bits into  $m = 2^p$  registers; in register  $r$  keep  $R[r] = \max$  leading-zero run-length seen in the remaining bits. Estimate  $\hat{D}$  by the standard harmonic-mean, bias-corrected estimator; space  $O(m)$ , update  $O(1)$ , relative error  $\approx 1.04/\sqrt{m}$ .

```

Algorithm: hll_update
Input: registers R[0..m-1], hash h, item x
Output: updated R

w ← h(x) // 64-bit
r ← first p bits of w as integer in [0..m-1]
z ← 1 + number_of_leading_zeros( w >> p )
R[r] ← max(R[r], z)
// Estimation done periodically from R by harmonic mean formula

```

✓ **Answer:** Use HLL (or FM) to achieve  $O(1)$  updates with small, controllable error; exact counting requires a large hash set.

### Exercise 3 — —

**Assignment Source:** kt.pdf p.1

**Text Source:** slides

Design a multi-set dictionary with chaining; each node stores (key, count) .

```

Algorithm: multiset_add
Input: array A of lists, key x, hash h
Output: increment count of x

i ← h(x)
for (k,c) in A[i]:
    if k = x: c ← c+1; return
prepend (x,1) to A[i]

Algorithm: multiset_remove
Input: A, x, h

i ← h(x)
for (k,c) in A[i]:
    if k = x:
        if c>1: c ← c-1 else remove node
    return

Algorithm: multiset_report
Input: A, x, h

```

```

i ← h(x)
for (k,c) in A[i]: if k=x: return c
return 0
// Expected time  $O(1)$  each; space  $O(m + \text{\#distinct})$ 

```

✓ **Answer:** Expected  $O(1)$  ADD/REMOVE/REPORT with chaining and per-key counters.

---

#### Exercise 4.1 — —

**Assignment Source:** kt.pdf p.1

**Text Source:** slides

**Lazy deletion** in linear probing. Use a special marker  $\otimes$  (tombstone). - **SEARCH:** treat  $\otimes$  as **occupied** and continue scanning; stop only at a truly empty slot  $\emptyset$  or at the key. - **INSERT:** keys may reuse  $\otimes$ ; remember the first  $\otimes$  seen and place the new key there (or at the first  $\emptyset$  if no  $\otimes$  encountered).

✓ **Answer:** SEARCH unchanged except that  $\otimes$  does not terminate the scan; INSERT prefers the first tombstone encountered.

---

#### Exercise 4.2 — —

**Assignment Source:** kt.pdf p.1

**Text Source:** slides

**Pros/cons of lazy deletion.** - **Benefits:**  $O(1)$  deletion; preserves cluster invariants; avoids costly re-insert cascades. - **Drawbacks:** tombstones accumulate  $\rightarrow$  longer searches and inserts; perform periodic **rebuids** (rehash into a fresh table) when the tombstone fraction is high.

✓ **Answer:** Prefer lazy deletion for fast deletes; schedule rebuids to bound probe lengths.

---

#### Exercise 5 — KT §13 (Quicksort)

**Assignment Source:** kt.pdf p.1

**Text Source:** KT Ch.13

**Claim.** Insert a random permutation into an empty BST; output the inorder traversal. The expected running time is  $O(n \log n)$ .

**Reasoning.** The BST shape from random insertion mirrors Quicksort with random pivots. Each key acts as a pivot once; expected split quality gives  $O(n \log n)$  comparisons even though the BST does not rebalance. (KT §13.)

✓ **Answer:**  $O(n \log n)$  expected time; this process is an alternative description of randomized Quicksort.

---

### Exercise 6.1 — —

**Assignment Source:** kt.pdf p.2

**Text Source:** slides; KT §13.6

When  $n \gg m$ , the load factor  $\alpha = n/m$  is large. With chaining: time per operation  $O(1 + \alpha) = \Theta(n/m)$ ; space  $O(m + n) = \Theta(n)$  dominated by elements. With open addressing: probe sequences become long; costs blow up as  $\alpha \rightarrow 1$ .

✓ **Answer:** Performance degrades linearly with  $\alpha$ ; increase  $m$  (resize) to restore constant expected time.

---

### Exercise 6.2 — —

**Assignment Source:** kt.pdf p.2

**Text Source:** slides; KT §13.6

**Growth-handling.** Maintain  $\alpha \leq \alpha_{\max}$  (e.g., 0.7). On INSERT that makes  $\alpha > \alpha_{\max}$ , set  $m \leftarrow 2m$  and **rehash** all keys; optionally halve when  $\alpha < \alpha_{\min}$ .

```
Algorithm: dict_insert_with_resize
Input: table A, counts (n, m), thresholds  $\alpha_{\max}$ ,  $\alpha_{\min}$ , hash family H
Output: A with x inserted; amortized  $O(1)$ 

if (n+1)/m >  $\alpha_{\max}$ :
    m  $\leftarrow$  2m
    choose new h  $\in$  H; allocate A'[0..m-1] empty
    for each key y in A: lp_insert_or_chain(A', y, h)
    A  $\leftarrow$  A'
insert x into A using h; n  $\leftarrow$  n+1
// Time: amortized expected  $O(1)$ ; Space:  $O(m)$ 
```

✓ **Answer:** Doubling/rehashing (and optional halving) yields compact space and expected amortized  $O(1)$  per operation.

---

### Exercise 7.1 — —

**Assignment Source:** kt.pdf p.2

**Text Source:** slides

Let  $b$  be bushes and  $k$  carrots hidden in  $k$  distinct bushes. Each round Billy picks a bush uniformly at random.

The event “finds a carrot” has probability  $p = k/b$ . Trials to first success are geometric with mean  $1/p = b/k$ .

✓ **Answer:** Expected bushes before the first carrot:  $\mathbf{E} = b/k$ .

---

## Exercise 7.2 — —

**Assignment Source:** kt.pdf p.2

**Text Source:** slides

He continues until he has found three distinct carrots. After  $j$  carrots already found, success probability is  $(k - j)/b$ . The additional trials needed are geometric with mean  $b/(k - j)$ .

Hence  $\mathbb{E}[T_3] = b\left(\frac{1}{k} + \frac{1}{k-1} + \frac{1}{k-2}\right)$ .

✓ **Answer:**  $\mathbb{E}[T_3] = b\left(\frac{1}{k} + \frac{1}{k-1} + \frac{1}{k-2}\right)$  (valid for  $k \geq 3$ ).

---

## Puzzle

Hash 64-bit integers into  $m = 2^{20}$  slots. Choose a universal family and give explicit parameters.

**One solution.** Pick odd  $a \in \{1, 3, 5, \dots, 2^{64} - 1\}$  and  $b \in [0, 2^{64} - 1]$  uniformly; let  $h_{a,b}(x) = ((a \cdot x + b) \bmod 2^{64}) \gg (64 - 20)$  (take the top 20 bits). This is the multiply-shift scheme; it forms a universal family and is extremely fast.

✓ **Answer:** Any concrete  $a, b$  as above (e.g.,  $a = 0x9E3779B97F4A7C15$  ,  $b = 0xD1B54A32D192ED03$  ) defines a valid  $h$ .

---

## Summary

- Use chaining or open addressing with a **good** hash family; prefer universal hashing to de-correlate inputs.
- Keep load factor bounded via **resizing** to ensure expected  $O(1)$  operations.
- For streams, approximate distinct counters (HLL/FM) provide  $O(1)$  updates with small memory.
- Linear probing is cache-friendly; combine with tombstones and periodic rebuilds.
- Randomized BST-sort equals Quicksort in expectation  $\rightarrow O(n \log n)$ .

**Notation recap.**  $n$  items, table size  $m$  , load factor  $\alpha = n/m$  ,  $h : U \rightarrow [0, m - 1]$  ,  $\emptyset$  empty,  $\otimes$  tombstone.