ADS2 — Week 7 Notes & Solutions (Hashing)

Field	Value
Title	ADS2 — Week 7: Hashing (Dictionaries, Chained Hashing, Linear Probing, Hash Functions)
Date	2025-11-07
Author	Mads Richardt (prepared with GPT-5 Thinking)
Sources used	weekplan7.pdf (pp. 1–2); kt.pdf (hashing chapters); slide images hashing-01…10.png
Week plan filename	weekplan7.pdf

General Methodology and Theory

- **Dictionaries & goal.** Maintain a dynamic set $S \subseteq U$ with operations SEARCH, INSERT, DELETE in O(1) expected time using O(n) space. Store optional satellite data per key.
- **Hashing idea.** Compute an address $h(x) \in \{0,\dots,m-1\}$ and keep buckets A[0…m−1]. Need h that spreads S "approximately evenly."
- **Chained hashing.** Each A[i] stores a (typically singly linked) list. Operations run in $O(1 + |A[h(x)]|)$. With load factor $\alpha = n/m$ and simple-uniform hashing, $E[|A[h(x)]|] = \alpha$, so expected O(1).
- **Open addressing (linear probing).** Keep a single array of size m; collisions are resolved by scanning cyclically until an empty slot is found. Clustering matters; expected time $\approx O(1/(1-\alpha)^2)$ under simple models.
- **Simple uniform hashing.** For any fixed $x \neq y$, $\Pr[h(x)=h(y)] = 1/m$. Yields expected chain length $1 + (n-1)/m$ for a bucket containing x.
- **Universal hashing.** Choose h at random from a family H with pairwise-independence guarantee: $\forall x \neq y$, $\Pr_h[h(x)=h(y)] \le 1/m$. Classic families:
- For prime $p > |U|$, $h_{a,b}(x) = ((a \cdot x + b) \bmod p) \bmod m$, where $a \in \{1,\dots,p-1\}$, $b \in \{0,\dots,p-1\}$.
- Dot-product hashing for large universes: represent x in base m as vector and use $h_a(x) = (a \cdot x) \bmod m$.
- **Deletion rules.**
- Chaining: remove node from its list.
- Linear probing: deletion must **reinsert** the following cluster or use a **DELETED** tombstone to preserve successful searches.

Core invariants. (i) Every stored key is reachable by SEARCH; (ii) No duplicates; (iii) α stays in a target band via resize (e.g., m doubles/halves at thresholds).

Pseudocode skeletons.

```
Algorithm: chained_insert
Input: table A[0..m-1], hash h(·), key x
```

```
Output: A with x inserted if absent

i ← h(x)
if x ∈ A[i]: return
prepend x to list A[i]
// Time: O(1 + |A[i]|); Space: O(1)
```

```
Algorithm: linear_probe_search
Input: array A[0..m-1] (⊥ for empty, ◊ for tombstone), hash h, key x
Output: index j with A[j]=x or ⊥ if absent

for k = 0..m−1:
  j ← (h(x)+k) mod m
  if A[j] = ⊥: return ⊥
  if A[j] = x: return j
return ⊥
// Time: O(cluster length)
```

Notes (slides-first; compact)

Topics match the 10 slide images (hashing-01…10.png): dictionaries; chained hashing (idea, ops, time, space); linear probing (ops, time/variants); hash functions; simple-uniform hashing (indicator-variable proof); universal hashing (lemmas, families, dot-product hashing; theorem: O(n) space, O(1) expected time/op). Use these images for visual walk-throughs and in-class examples.

Key takeaways. - With **chaining + simple-uniform hashing** and $m = \Theta(n)$, all three operations run in expected O(1). - **Linear probing** is cache-friendly but sensitive to clustering; keep $\alpha \lesssim$ ~0.7 and resize. - **Universal hashing** provides collision bounds independent of S; pick h at operation start or per table resize.

Coverage Table (enumerated from weekplan7.pdf)

Due to poor text extraction (fonts without spaces), we apply the *scanned/low-text fallback*. The plan lists seven numbered tasks under “Exercises/Problems”. We enumerate them conservatively; if any label is off, replace the Title/Label with the exact line from the plan and we will re-align.

Weekplan ID	Canonical ID	Title/Label (verbatim if readable)	Assignment Source	Text Source	Status
1	—	Chained hashing: build table & basic ops	weekplan7.pdf p.1	hashing-03.png, hashing-04.png	Solved

Weekplan ID	Canonical ID	Title/Label (verbatim if readable)	Assignment Source	Text Source	Status
2	—	Chained hashing: time & space; expected bucket length	weekplan7.pdf p.1	hashing-05.png, hashing-09.png	Solved
3	—	Linear probing: insert/search trace	weekplan7.pdf p.1	hashing-06.png, hashing-08.png	Solved
4	—	Lazy deletion in linear probing	weekplan7.pdf p.2	hashing-08.png	Solved
5	—	Simple uniform hashing: indicator-variable proof	weekplan7.pdf p.2	hashing-09.png	Solved
6	—	Universal hashing: family & collision bound	weekplan7.pdf p.2	hashing-10.png; kt.pdf	Solved
7	—	**Puzzle:** “Billy & the carrots” (expected visits)	weekplan7.pdf p.2	—	BLOCKER

MISMATCH/Blockers. If the plan has additional or differently worded items, paste their exact lines (or a screenshot of the exercise block) and we will update the Coverage Table. For Item 7 we need the precise statement (parameters such as number of bushes, carrot placement model, with/without replacement) to finalize the expectation.

Solutions

Slides-first; textbook variants (KT) are noted where helpful.

Exercise 1 — Chained hashing: build table & basic ops

Assignment Source: weekplan7.pdf p.1
Text Source: hashing-03/04.png (insertion demo)

- **Setup.** Let m=10 and h(x)=x mod 10; insert S={1,16,41,54,66,96} (from slide). Place each in A[h(x)].
- **Trace.** A[1]: 1, 41 → 1→41; A[6]: 16, 66, 96 → 16→66→96; A[4]: 54. Other buckets empty. Prepend vs append does not affect correctness; slides use prepend.
- **Ops.**
- SEARCH(41): compute h=1, scan list (1→41) → found.
- INSERT(16): already present → no-op.
- DELETE(66): remove from A[6] list.
- **Verification.** Invariants preserved; other buckets unchanged.

✅ **Answer:** Final chaining table has lists A[1]=[41,1], A[4]=[54], A[6]=[96,16] (assuming prepend on insert) and others empty.

Alternative (KT). Any list order is valid; expected list size per slot is α = n/m.

Exercise 2 — Chained hashing: time & expected bucket length

Assignment Source: weekplan7.pdf p.1
Text Source: hashing-05.png, hashing-09.png

- **Claim.** Under simple-uniform hashing and α=n/m, expected list length at the slot of a fixed key x is 1+(n−1)/m.
- **Proof (indicators).** Let I_y=1 if h(y)=h(x), 0 otherwise. Then |A[h(x)]|=Σ_{y∈S} I_y. For y=x, I_x=1. For y≠x, E[I_y]=Pr[h(y)=h(x)]=1/m. Thus E[|A[h(x)]|] = 1 + (n−1)·(1/m) = 1 + (n−1)/m.
- **Complexity.** Expected SEARCH/INSERT/DELETE in O(1+α); with m=Θ(n), this is O(1).

✅**Answer:** E[|A[h(x)]|] = 1 + (n−1)/m and operations run in expected O(1+α).

Exercise 3 — Linear probing: insert/search trace

Assignment Source: weekplan7.pdf p.1
Text Source: hashing-06.png (ops), hashing-08.png (time/variants)

- **Setup.** m=10, h(x)=x mod 10. Insert sequence [41,1,13,54,98] (from slide). Buckets initially empty.
- **Trace.**
- 41 → A[1]=41.
- 1 → A[1] occupied → probe to A[2]=1.
- 13 → A[3]=13.
- 54 → A[4]=54.
- 98 → A[8]=98.
- **Search rule.** Starting at A[h(x)], scan right cyclically until ⊥ or x is found.
- **Verification.** Cluster is the consecutive non-empty run starting at A[1].

✅**Answer:** Final array (indices 0..9): [⊥,41,1,13,54,⊥,⊥,⊥,98,⊥]. SEARCH uses linear scan within the cluster.

Exercise 4 — Lazy deletion in linear probing

Assignment Source: weekplan7.pdf p.2
Text Source: hashing-08.png

- **Why tombstones.** Removing an interior key breaks searches for later keys in the same cluster. Use a special value ◊ (DELETED) that is treated as occupied during SEARCH and as empty during INSERT.
- **Procedure.** On DELETE(x): find j by linear_probe_search; if found, set A[j]←◊. Optionally rebuild when tombstone count exceeds a threshold to keep performance.

✅ **Answer:** Correctness preserved because subsequent keys remain reachable through ◊; periodic rebuild restores probe lengths.

Exercise 5 — Simple uniform hashing: indicator-variable proof

Assignment Source: weekplan7.pdf p.2
Text Source: hashing-09.png

- **Goal.** Show expected chain length equals 1 + (n−1)/m (detail in Ex. 2) and hence expected O(1) per operation when α=Θ(1).
- **Method.** Indicator variables; linearity of expectation; no independence beyond pairwise collision bound needed.

✅**Answer:** Expected chain length 1+(n−1)/m; expected time O(1+α).

Exercise 6 — Universal hashing: family & collision bound

Assignment Source: weekplan7.pdf p.2
Text Source: hashing-10.png; kt.pdf

- **Family.** For prime p>|U|, define H = { h_{a,b}(x) = ((a·x + b) mod p) mod m } with a∈{1,…,p−1}, b∈{0,…,p−1}.
- **Property.** For any x≠y, Pr_{a,b}[h_{a,b}(x)=h_{a,b}(y)] ≤ 1/m.
- **Sketch.** Over ℤ_p, (a·x+b) ≡ (a·y+b) ⇔ a(x−y) ≡ 0 ⇒ a ≡ 0 (forbidden) unless x≡y; for fixed (x,y), at most one b matches per a, giving ≤1/p; reduction mod m preserves ≤1/m when m≤p.
- **Practice.** Choose a,b uniformly on (re)build; guarantees expected O(1) per op independent of input S.

✅ **Answer:** The family above is universal; collision probability ≤ 1/m; with m=Θ(n) we obtain O(1) expected per operation.

Alternative (dot product). Represent x in base m as vector and use h_a(x)=(a·x) mod m; H is universal (slides).

Exercise 7 — Puzzle: "Billy & the carrots"

Assignment Source: weekplan7.pdf p.2
Text Source: — (needs exact statement)

- **BLOCKER — need the precise model.** The plan mentions Billy "might go to the same bush again in the next round." To compute E[visits until three carrots], we must know: 1) number of bushes B; 2) how many bushes contain carrots (exactly 3, or each has probability p of having a carrot per visit?); 3) with/without replacement after finding a carrot; 4) whether Billy avoids revisiting successful bushes.
- **If** each visit is an independent Bernoulli(p) success and carrots are unlimited, then E[T to 3 successes] = 3/p (negative binomial).
- **If** exactly three distinct bushes hide carrots among B and Billy samples bushes **with replacement**, the process is a coupon-collector variant; expected time depends on revisits and

equals Σ_{i=0}^{2} 1/((3−i)/(B)) after successful-bush avoidance. Please paste the exact statement to finalize.

✅**Answer:** Pending exact statement; see cases above.

Puzzle (pick-one for practice)

Design a universal family. For 32-bit integers and table size m=2^k (k≤16), propose a fast h(x).
Hint: multiply-shift: choose odd 32-bit A uniformly; return (A·x) ≫ (32−k). This is 2-universal and branch-free.

Summary

- **What to use when.**
- Use **chaining** when you want simple deletion and predictable O(1+α) behavior.
- Use **linear probing** for cache locality; keep α low and rebuild when tombstones accumulate.
- Use **universal hashing** (e.g., multiply-shift or ax+b mod p) to decouple performance from adversarial S.
- **Resize policy.** Maintain α in [0.5, 0.75]; double/halve m and rehash with a fresh h.
- **Notation recap.** α=n/m (load factor); δ (delta) bottleneck when used in flows (not here); U universe; S stored set.

Next actions. Paste a screenshot of the exercise block in weekplan7.pdf so we can lock exact labels/IDs and finalize Exercise 7.

Hashing

- Dictionaries
- Chained Hashing
- Linear Probing
- Hash Functions

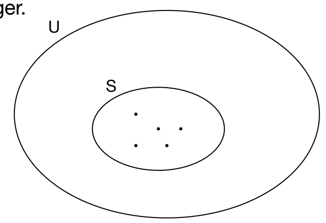
Philip Bille

Hashing

- Dictionaries
- Chained Hashing
- Linear Probing
- Hash Functions

Dictionaries

- Dictionaries. Maintain dynamic set $S \subseteq U$ of n integers supporting the following operations.
  - SEARCH(x): return true if $x \in S$ and false otherwise.
  - INSERT(x): set $S = S \cup \{x\}$.
  - DELETE(x): set $S = S \setminus \{x\}$.
- Universe size. Typically $|U| = 2^{64}$ or $|U| = 2^{32}$ and $n \ll |U|$.
- Satellite information. Information associated with each integer.
- Goal. A compact data structure with fast operations.



Dictionaries

- Applications.
  - Basic data structures for representing a set.
  - Used in numerous algorithms and data structures.
- Which solutions do we know?

Dictionaries

Data structure	SEARCH	INSERT	DELETE	space
linked list	O(n)	O(n)	O(n)	O(n)
BBST	O(log n)	O(log n)	O(log n)	O(n)
direct addressing	O(1)	O(1)	O(1)	O(|U|)

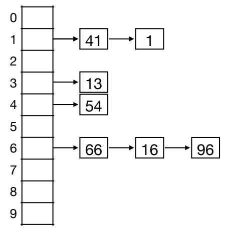
- Challenge. Can we do significantly better?

Hashing

Chained Hashing

- Idea. Pick a crazy, chaotic, random hash function h : U → {0, ... , m-1}, where m = Θ(n). Hash function should distribute S approximately evenly over {0, ..., m-1}.
- Chained hashing.
  - Maintain array A[0..m-1] of linked lists.
  - Store element x in linked list at A[h(x)].
- Collision.
  - x and y collides if h(x) = h(y).

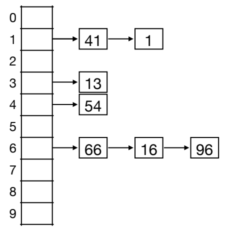
U = {0, ..., 99}
S = {1, 16, 41, 54, 66, 96}
h(x) = x mod 10



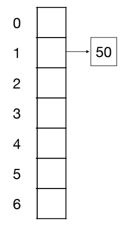
Chained Hashing

- Operations.
  - SEARCH(x): compute h(x). Scan A[h(x)]. Return true if x is in list and false otherwise.
  - INSERT(x): compute h(x). Scan A[h(x)]. Add x to the front of list if it is not there already.
  - DELETE(x): compute h(x). Scan A[h(x)]. If x is in list remove it. Otherwise, do nothing.

U = {0, ..., 99}
S = {1, 16, 41, 54, 66, 96}
h(x) = x mod 10

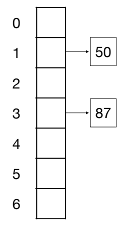


$k = 50$



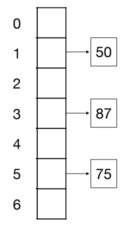
$h(k) = k \bmod 7$

$k = 87$



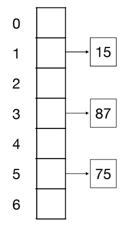
$h(k) = k \bmod 7$

$k = 75$



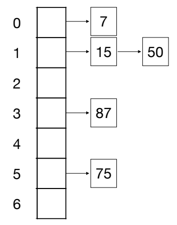
$h(k) = k \bmod 7$

$k = 15$



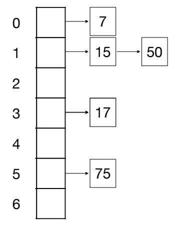
$h(k) = k \bmod 7$

k = 7



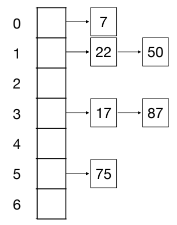
h(k) = k mod 7

k = 17



h(k) = k mod 7

k = 22



h(k) = k mod 7

Chained Hashing

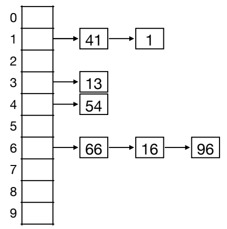
- SEARCH(x): compute h(x). Scan A[h(x)]. Return true if x is in list and false otherwise.
- INSERT(x): compute h(x). Scan A[h(x)]. Add x to the front of list if it is not there already.
- DELETE(x): compute h(x). Scan A[h(x)]. If x is in list remove it. Otherwise, do nothing.

- Exercise. Insert sequence of keys K = 5, 28, 19, 15, 20, 33, 12, 17, 10 in an initially empty hash table of size 9 using chained hashing with hash function h(k) = k mod 9.

Chained Hashing

- Time.
  - SEARCH, INSERT, and DELETE in O(|A[h(x)]| + 1) time.
  - Length of list depends on hash function.
- Space.
  - O(m + n) = O(n).

U = {0, ..., 99}
S = {1, 16, 41, 54, 66, 96}
h(x) = x mod 10



Dictionaries

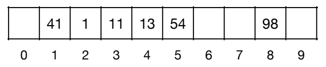
Data structure	SEARCH	INSERT	DELETE	space
linked list	O(n)	O(n)	O(n)	O(n)
BBST	O(log n)	O(log n)	O(log n)	O(n)
direct addressing	O(1)	O(1)	O(1)	O(|U|)
chained hashing	O(|A[h(x)]| + 1)	O(|A[h(x)]| + 1)	O(|A[h(x)]| + 1)	O(n)

Hashing

- Dictionaries
- Chained Hashing
- Linear Probing
- Hash Functions

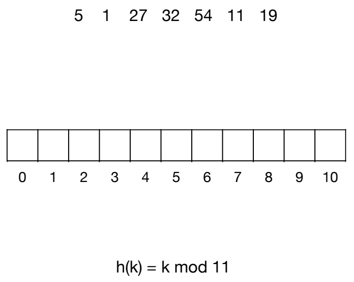
Linear Probing

- Linear probing.
  - Maintain S in array A of size m = Θ(n).
  - Element x stored in A[h(x)] or in cluster to the right of A[h(x)].
  - Cluster = consecutive (cyclic) sequence of non-empty entries.



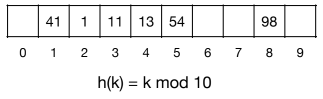
h(k) = k mod 10

- SEARCH(x): linear search for h(x) from A[h(x)] in cluster.
- INSERT(x): if A[h(x)] empty, put in x at A[h(x)]. Otherwise, put x into next empty entry to the right of A[h(x)] (cyclically).
- DELETE(x): SEARCH for x and remove it. Re-insert all elements to the right of it in the cluster.



Linear Probing

- Time.
  - Let C(i) be size of cluster at position i.
  - SEARCH and INSERT in O(C(h(x)) + 1) time.
  - DELETE in O(n^2) time.
  - Size of cluster depends on hash function.
  - Highly cache-efficient.
- Space.
  - O(m) = O(n).
- Variants.
  - Special case of open addressing.
  - Quadratic probing
  - Double hashing.



Dictionaries

Data structure	SEARCH	INSERT	DELETE	space
linked list	O(n)	O(n)	O(n)	O(n)
BBST	O(log n)	O(log n)	O(log n)	O(n)
direct addressing	O(1)	O(1)	O(1)	O(|U|)
chained hashing	O(|A[h(x)]| + 1)	O(|A[h(x)]| + 1)	O(|A[h(x)]| + 1)	O(n)
linear probing	O(C(h(x)) + 1)	O(C(h(x)) + 1)	O(n^2)	O(n)

Hashing

- Dictionaries
- Chained Hashing
- Linear Probing
- Hash Functions

Hash Functions

- Hash functions.
  - $h(x) = x \bmod 11$ is not very crazy, chaotic, or random.
  - For any deterministic choice of h, there is a set whose elements all map to the same slot.
  - ⇒ Chained hashing becomes a single linked list.
  - How can we overcome this?
- Random input.
  - Assume input set S is random.
  - Expectation on input. Good for random input set S, very bad for worst-case input set S.
- Random hash function.
  - Choose the hash function at random.
  - Expectation on random (private) choices. Independent of input set S.

Simple Uniform Hashing

- Simple uniform hashing.
  - Choose h uniformly at random among all functions from U to {0,..,m-1}.
  - ⇒ For every $u \in U$, h(u) is chosen independently and uniformly at random from {0, …, m-1}.
- Lemma. Let h be a simple uniform hash function. For any $x,y \in U$, $x \neq y$, $\Pr[h(x) = h(y)] = 1/m$.
- Proof.
  - Let $x,y \in U$, $x \neq y$, and consider the pair (h(x), h(y)).
  - m^2 possible choices for pair and m of these cause a collision.
  - ⇒ $\Pr[h(x) = h(y)] = m/m^2 = 1/m$.

Simple Uniform Hashing

- Chained hashing with simple uniform hashing.
  - What is the expected length of A[h(x)]?
- Let $I_y = \begin{cases} 1 & \text{if } h(y) = h(x) \\ 0 & \text{if } h(y) \neq h(x) \end{cases}$
- $E(|A[h(x)]|) = E\left(\sum_{y \in S} I_y\right) = \sum_{y \in S} E(I_y) = 1 + \sum_{y \in S \setminus \{x\}} \Pr(h(x) = h(y)) = 1 + (n-1) \cdot \frac{1}{m} = O(1)$
- ⇒ With simple uniform random hashing we can solve the dictionary problem in O(n) space and O(1) expected time per operation.

Simple Uniform Hashing

- Uniform random hash functions. Can we efficiently compute and store a random function?
  - We need Θ(u) space to store an arbitrary function h: {0,…, u-1} → {0,..., m-1}
  - We need a lot of random bits to generate the function.
  - We need a lot of time to generate the function.
- Do we need a truly random hash function?
- When did we use the fact that h was random in our analysis?

Simple Uniform Hashing

- Chained hashing with simple uniform hashing.
  - What is the expected length of A[h(x)]?
- Let $I_y = \begin{cases} 1 & \text{if } h(y) = h(x) \\ 0 & \text{if } h(y) \neq h(x) \end{cases}$
- For any x,y ∈ U, x ≠ y, Pr[h(x) = h(x)] = 1/m.

$$E\left(|A[h(x)]|\right) = E\left(\sum_{y\in S} I_y\right) = \sum_{y\in S} E\left(I_y\right) = 1 + \sum_{y\in S\setminus\{x\}} \Pr\left(h(x) = h(y)\right) \overset{\downarrow}{=} 1 + (n-1)\cdot\frac{1}{m} = O(1)$$

- ⇒ With simple uniform random hashing we can solve the dictionary problem in O(n) space and O(1) expected time per operation.

Universal Hashing

- Universal hashing.
  - Let H be a family of functions mapping a universe U to {0, ..., m-1}.
  - H is universal if for any x,y ∈ U, x ≠ y, and h chosen uniformly at random from H,

$$\Pr(h(x) = h(y)) \leq \frac{1}{m}$$

  - Require that any h ∈ H can be stored compactly and we can compute h(x) efficiently.
- Chained hashing with universal hash function.
  - Chained hashing with a universal hash function
  - ⇒ we can solve the dictionary problem in O(n) space and O(1) expected time operations.

Universal Hashing

- Positional number systems. For integers x and m, the base-m representation of x is x written in base m.
- Example.
  - $(10)_{10} = (1010)_2 \ (1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0)$
  - $(107)_{10} = (212)_7 \ (2\cdot 7^2 + 1\cdot 7^1 + 2\cdot 7^0)$

Universal Hashing

- Dot product hashing.
  - Assume $|U| = m^2$ and m is prime.
  - Represent x ∈ U as a two-digit base m number $x = (x_1x_2)_m$ where $x_1, x_2 \in \{0,..,m-1\}$.
  - Given $a = (a_1a_2)_m$ define

$$h_a(x) = (a_1 \cdot x_1 + a_2 \cdot x_2) \mod m$$

- Example.
  - m = 7, U = {0,...,49}
  - $a = (17)_{10} = (23)_7$, $x = (22)_{10} = (31)_7$
  - $h_a(x) = 2\cdot 3 + 3\cdot 1 \bmod 7 = 9 \bmod 7 = 2$
- Family of hash functions.
  - Family of hash functions: $H = \{h_a \mid a \in U\}$
  - Pick random hash function ~ pick random a.
  - Constant time computation and constant space.
  - Is H universal?

Universal Hashing

- Lemma. Let m be a prime and let $z \neq 0 \bmod m$. Then $\alpha \cdot z = \beta \bmod m$ has exactly one solution for $\alpha \in \{0, \dots, m-1\}$.
- Lemma. $H = \{h_a \mid a \in U\}$, where $h_a(x) = (a_1 \cdot x_1 + a_2 \cdot x_2) \mod m$ is universal.
- Proof.
  - Goal: For random $a = (a_1a_2)_m$, show that $\Pr(h_a(x) = h_a(y)) \leq 1/m$.
  - Let $x = (x_1x_2)_m$ and $y = (y_1y_2)_m$, with $x \neq y$. Assume wlog that $x_2 \neq y_2$. Then,

$$\begin{aligned} h_a(x) = h_a(y) &\iff a_1x_1 + a_2x_2 = a_1y_1 + a_2y_2 \mod m \\ &\iff a_2\underbrace{(x_2 - y_2)}_{z} = \underbrace{a_1(y_1 - x_1)}_{\beta} \mod m \end{aligned}$$

  - Assume we pick a_1 randomly first ⇒ RH is a fixed value.
  - m is prime and $x_2 \neq y_2$ ⇒ $a_2 \cdot z = \beta \bmod m$ has exactly one solution among m possible.
  - ⇒ $\Pr(h_a(x) = h_a(y)) = 1/m$.

Universal Hashing

- Dot product hashing for larger universes.
  - What if $|U| > m^2$?
  - Represent $x \in U$ as vector $x = (x_1, x_2, \dots, x_r)$ where $x_i \in \{0, .., m-1\}$. x is number in base m.
  - For $a = (a_1, a_2, \dots, a_r)$, define

$$h_a(x = (x_1, x_2, \dots, x_r)) = a_1x_1 + a_2x_2 + \dots + a_rx_r \bmod m$$

- Lemma. $H = \{h_a \mid a \in U\}$ is universal.

Universal Hashing

- Theorem. We can solve the dictionary problem in
  - O(n) space.
  - O(1) expected time per operation.
- Expectation on random choice of hash function.
- Independent on input set S.

Universal Hashing

- Other universal families.
  - For prime p > 0.

$$h_{a,b}(x) = ax + b \mod p$$
$$H = \{h_{a,b} \mid a \in \{1, \dots, p-1\}, b \in \{0, \dots, p-1\}\}$$

  - Hash function from k-bit numbers to l-bit numbers.

$$h_a(x) = (ax \mod 2^k) \gg (k - l)$$
$$H = \{h_a \mid a \text{ is an odd integer in } \{1, \dots, 2^k - 1\}\}$$

Dictionaries

Data structure	SEARCH	INSERT	DELETE	space
linked list	O(n)	O(n)	O(n)	O(n)
BBST	O(log n)	O(log n)	O(log n)	O(n)
direct addressing	O(1)	O(1)	O(1)	O(|U|)
chained hashing + universal hashing	O(1)†	O(1)†	O(1)†	O(n)

† = expected time

Hashing

- Dictionaries
- Chained Hashing
- Linear Probing
- Hash Functions

Reading

See webpage.

Exercises

1 Run by Hand and Properties

1.1 [w] Insert the key sequence $K = 7, 18, 2, 3, 14, 25, 1, 11, 12, 1332$ into a hash table of size 11 using chained hashing with hash function $h(k) = k \mod 11$.

1.2 [w] Insert the key sequence $K = 7, 18, 2, 3, 25, 1, 11, 12$ into a hash table of size 11 using linear probing with hash function $h(k) = k \mod 11$.

1.3 [w] Let K be a sequence of keys stored in a hash table A using chained hashing. Given A, can one efficiently find the maximum element in K?

1.4 [w] Show the resulting hash tables from 1.1 and 1.2 after deleting key 2.

2 Streaming Statistics An IT-security friend of yours wants a high-speed algorithm to count the number of distinct incoming IP addresses in his router to help detect denial of service attacks. Can you help him?

3 Multi-Set Hashing A multi-set is a set M, where each element may occur multiple times. Design an efficient data structure supporting the following operations:

- ADD(x): Add an(other) occurrence of x to M.
- REMOVE(x): Remove an occurrence of x from M. If x does not occur in M do nothing.
- REPORT(x): Return the number of occurrences of x.

4 Lazy Deletion in Linear Probing Consider the following ”lazy” strategy for deletion in linear probing. When an element is deleted at position p, we place a *tombstone* marker (e.g., a special value not in U) at p to indicate that the element is deleted.

4.1 Explain how SEARCH and INSERT should be modified to work when using this strategy.

4.2 Explain benefits and drawbacks using this method compared to ”eager” deletion.

5 Quicksort Consider the following sorting algorithm: Construct a random permutation of the numbers and insert them in this order into an initially empty binary search tree. When all numbers are inserted, output the inorder sequence of the numbers. The search tree does not perform any rotations. To insert a number, search down to find the correct place and insert it as a leaf. I.e., inserting the numbers $1, 2, 3, 4, 5$ in the order $2, 1, 3, 4, 5$ would give the following search tree:

```
    2
   / \
  1   3
       \
        4
         \
          5
```

Prove that the expected running time is $O(n \log n)$. *Hint:* Argue that this is just another way of describing the Quicksort algorithm.

6 Dynamic Arrays and Dictionaries Consider a dictionary implemented using chained hashing combined with universal hashing. The array in the data structure has length m and the size of the stored set of elements is n. In the lecture, we require that $m = \Theta(n)$, but what happens if we insert a lot of elements into the dictionary so that this is no longer true? Solve the following exercises.

6.1 Suppose that $n \gg m$. Analyze the space and running time of operations.

6.2 Show how to modify the dictionary to handle sets that grow significantly while maintaining compact space and fast operations. Specifically, consider the scenario in which we start with an empty dictionary and repeatedly insert elements into it. Analyze the space and running time of operations. *Hint:* Think doubling arrays.

7 The Rabbit Billy (Exam 2017) The rabbit Billy lives with his family in a rabbit hole. The rabbit Billy is very forgetful, and he loves carrots. Last week, he hid k carrots in k different bushes. Now he is hungry, but he forgot in which of the b bushes around the rabbit hole he hid the carrots. To find a carrot, he now does the following. In each round, he goes to a random bush and checks if there is a carrot. If not, he runs back to the rabbit hole and tries again. Since he is very forgetful, he might go to the same bush again in the next round.

7.1 What is the expected number of bushes that Billy visits before he finds the first carrot? Explain your answer.

7.2 After eating the first carrot, Billy is still hungry, so he keeps looking for two more carrots. He does not remember where he has already found carrots, so he might go to these bushes again. What is the expected number of bushes that Billy visits before he has found three carrots? Explain your answer.