ADS2 — Hashing (Week Notes & Full Solutions)

Field	Value
Title	ADS2 — Hashing: Dictionaries, Chaining, Linear Probing, Universal Hashing
Date	2025-11-08
Author	Mads Richardt
Sources used	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
Week plan filename	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 mod 11)	kt.pdf p.1	slides; KT §13.6	Solved
1.2	—	Insert K into linear probing (m=11, h(k)=k mod 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 ≫ 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 α 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 α (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 → value):

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

✅**Answer:** As above ($\varnothing$ 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 $A[2] \to A[2]$ 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: $0 \to 11, 1 \to 1, 2 \to 12, 3 \to 3, 4 \to 14, 5 \to 25, 6 \to 1332, 7 \to 7, 8 \to 18, 9 \to \varnothing, 10 \to \varnothing$.

```
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 $h : \mathrm{IP} \to \{0,1\}^{64}$; 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 $(\mathrm{key}, \mathrm{count})$.

```
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 + #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 ⊗ (tombstone). - **SEARCH:** treat ⊗ as **occupied** and continue scanning; stop only at a truly empty slot ∅ or at the key. - **INSERT:** keys may reuse ⊗; remember the first ⊗ seen and place the new key there (or at the first ∅ if no ⊗ encountered).

✅**Answer:** SEARCH unchanged except that ⊗ 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 → longer searches and inserts; perform periodic **rebuilds** (rehash into a fresh table) when the tombstone fraction is high.

✅**Answer:** Prefer lazy deletion for fast deletes; schedule rebuilds 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 $\alpha \to 1$.

✅**Answer:** Performance degrades linearly with α ; 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 $\alpha \le \alpha_{\max}$ (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 α_max, α_min, hash family H
Output: A with x inserted; amortized O(1)

if (n+1)/m > α_max:
  m ← 2m
  choose new h ∈ H; allocate A'[0..m-1] empty
  for each key y in A: lp_insert_or_chain(A', y, h)
  A ← A'
insert x into A using h; n ← 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 $\mathbf{E}[T_3] = b\left(\frac{1}{k} + \frac{1}{k-1} + \frac{1}{k-2}\right)$.

✅**Answer:** $\mathbf{E}[T_3] = b\left(\frac{1}{k} + \frac{1}{k-1} + \frac{1}{k-2}\right)$ (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 $a \in \{1, 3, 5, \ldots, 2^{64}-1\}$ 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 $\to O(n \log n)$.

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

Hashing

- Dictionaries
- Chained Hashing
- Linear Probing
- Hash Functions

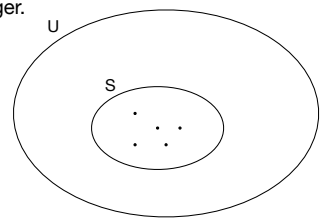
Philip Bille

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?

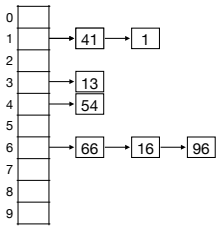
Hashing

- Dictionaries
- Chained Hashing
- Linear Probing
- Hash Functions

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).

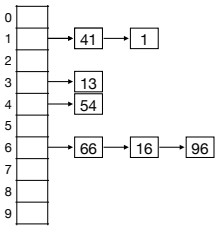
U = {0, ..., 99}
S = {1, 16, 41, 54, 66, 96}
h(x) = x mod 10



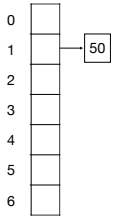
Chained Hashing

- 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

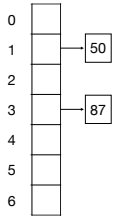


k = 50



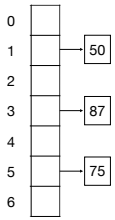
h(k) = k mod 7

k = 87



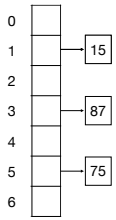
h(k) = k mod 7

k = 75



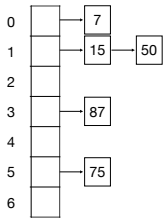
h(k) = k mod 7

k = 15



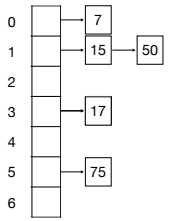
h(k) = k mod 7

k = 7



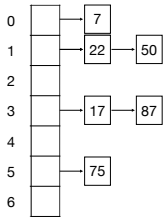
h(k) = k mod 7

k = 17



h(k) = k mod 7

k = 22



h(k) = k mod 7

Chained Hashing

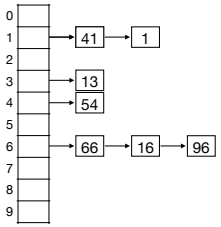
- SEARCH(x): compute h(x). Scan A[h(x)]. Return true if x is in list and false otherwise.
- INSERT(x): compute h(x). Scan A[h(x)]. Add x to the front of list if it is not there already.
- DELETE(x): compute h(x). Scan A[h(x)]. If x is in list remove it. Otherwise, do nothing.

- 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

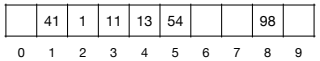
Data structure	SEARCH	INSERT	DELETE	space
linked list	O(n)	O(n)	O(n)	O(n)
BBST	O(log n)	O(log n)	O(log n)	O(n)
direct addressing	O(1)	O(1)	O(1)	O(|U|)
chained hashing	O(|A[h(x)]| + 1)	O(|A[h(x)]| + 1)	O(|A[h(x)]| + 1)	O(n)

Hashing

- Dictionaries
- Chained Hashing
- Linear Probing
- Hash Functions

Linear Probing

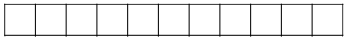
- Linear probing.
  - Maintain S in array A of size m = Θ(n).
  - Element x stored in A[h(x)] or in cluster to the right of A[h(x)].
  - Cluster = consecutive (cyclic) sequence of non-empty entries.



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.

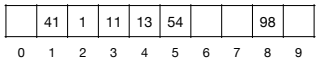
5 1 27 32 54 11 19



h(k) = k mod 11

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.



h(k) = k 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)
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 mod 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 ∈ 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 ∈ U, x ≠ y, Pr[h(x) = h(y)] = 1/m.
- Proof.
  - Let x,y ∈ U, x ≠ 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 $I_y = \begin{cases} 1 & \text{if } h(y) = h(x) \\ 0 & \text{if } h(y) \neq h(x) \end{cases}$
- $E\left(|A[h(x)]|\right) = E\left(\sum_{y \in S} I_y\right) = \sum_{y \in S} E\left(I_y\right) = 1 + \sum_{y \in S \setminus \{x\}} Pr\left(h(x) = h(y)\right) = 1 + (n-1) \cdot \frac{1}{m} = O(1)$
- ⇒ 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 $I_y = \begin{cases} 1 & \text{if } h(y) = h(x) \\ 0 & \text{if } h(y) \neq h(x) \end{cases}$

- $E(|A[h(x)]|) = E\left(\sum_{y \in S} I_y\right) = \sum_{y \in S} E(I_y) = 1 + \sum_{y \in S \setminus \{x\}} \Pr(h(x) = h(y)) \overset{\text{For any } x,y \in U,\ x \neq y,\ \Pr[h(x) = h(x)] = 1/m.}{\leq} 1 + (n-1) \cdot \frac{1}{m} = O(1)$

- ⇒ 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 $x \in U$ as a two-digit base m number $x = (x_1 x_2)_m$ where $x_1, x_2 \in \{0,..,m-1\}$.
  - Given $a = (a_1 a_2)_m$ 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 $\alpha \in \{0,..., m-1\}$.
- 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 $a = (a_1 a_2)_m$, show that $Pr(h_a(x) = h_a(y)) \leq 1/m$.
  - Let $x = (x_1 x_2)_m$ and $y = (y_1 y_2)_m$, with $x \neq y$. Assume wlog that $x_2 \neq y_2$. Then,

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

  - Assume we pick a_1 randomly first ⇒ RH is a fixed value.
  - m is prime and $x_2 \neq y_2 \Rightarrow a_2 \cdot z = \beta \bmod m$ 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 $x = (x_1, x_2,..., x_r)$ where $x_i \in \{0,..,m-1\}$. x is number in base m.
  - For $a = (a_1, a_2,\ldots, a_r)$, define

$$h_a(x = (x_1, x_2, \ldots, x_r)) = a_1 x_1 + a_2 x_2 + \ldots + a_r x_r \bmod m$$

- 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 $p > 0$.

$$h_{a,b}(x) = ax + b \mod p$$

$$H = \{h_{a,b} \mid a \in \{1, \ldots, p-1\}, b \in \{0, \ldots, p-1\}\}$$

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

$$h_a(x) = (ax \mod 2^k) \gg (k - l)$$

$$H = \{h_a \mid a \text{ is an odd integer in } \{1, \ldots, 2^k - 1\}\}$$

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:

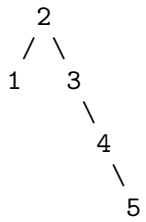
- ADD(x): Add an(other) occurrence of x to M.
- REMOVE(x): Remove an occurrence of x from M. If x does not occur in M do nothing.
- REPORT(x): Return the number of occurrences of x.

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:



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.