Hashing I

Recap: So far, ordered dicts.

- insert, delete, find
- Comparison-based: <, ==, >
- getMin, getMax, getK, findUp...
- Query/Update time: $O(\log n)$
  → Worst-case, amortized, random.

→ Can we do better? $O(1)$?

Hashing: (Unordered) dictionary

- stores key-value pairs in array table[0..m-1]
- supports basic dict. ops. (insert, delete, find) in $O(1)$ expected time
- does not support ordered ops (getMin, findUp, ...)
- simple, practical, widely used

Overview:

- To store n keys, our table should (ideally) be a bit larger (e.g., $m \geq c \cdot n$, $c = 1.25$)
- Load factor:

$$\lambda = n/m$$

- Running times increase as $\lambda \to 1$
- Hash function:

$$h : \text{Keys} \longrightarrow [0..m-1]$$

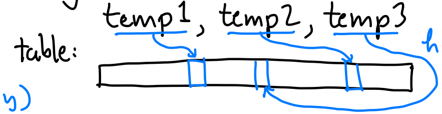
→ Should scatter keys random.

→ Need to handle collisions → $x \neq y$ but $h(x) = h(y)$

Good Hash Function:

- Efficient to compute
- Produce few collisions
  - Use every bit in key
  - Break up natural clusters

E.g. Java variable names: temp1, temp2, temp3



Common Examples:

Assume keys can be interpreted as ints

- Division hash:

$$h(x) = x \bmod m$$

- Multiplicative hash:

$$h(x) = (ax \bmod p) \bmod m$$

  a, p - large prime numbers

- Linear hash:

$$h(x) = ((ax + b) \bmod p) \bmod m$$

  a, b, p - large primes

Why "mod p mod m"?

- modding by a large prime scatters keys
- m may not be prime (e.g. power of 2)

Universal Hashing:

Even better → randomize!

- Let H be a family of hash fns
- Select $h \in H$ randomly
- If $x \neq y$ then $\text{Prob}(h(x) = h(y)) = 1/m$

E.g. Let p - large prime, $a \in [1..p-1]$, $b \in [0..p-1]$ all random

- $h_{a,b}(x) = ((ax + b) \bmod p) \bmod m$

Hashing II

Overview:

- Separate Chaining
- Open Addressing:
  - Linear probing
  - Quadratic probing
  - Double hashing

simple/slow ↓ complex/fast

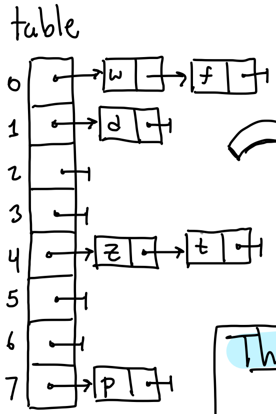
Separate Chaining:

table[i] is head of linked list of keys that hash to i.

Example:

Keys (x)	h(x)
d	1
z	4
p	7
w	0
t	4
f	0

m = 8



Analysis: Recall load factor

$\lambda = n/m$

n = # of keys

m = table size

Collision Resolution:

If there were no collisions hashing would be trivial!

insert(x,v) → table[h(x)] = v

find(x) → return table[h(x)]

delete(x) → table[h(x)] = null

S_{sc} = Expected search time if x found (successful)

U_{sc} = Expect. search time if x not found (unsuccessful)

Thm: $S_{sc} = 1 + \lambda/2$ $\quad U_{sc} = 1 + \lambda$

Proof: On avg. each list has $n/m = \lambda$

success: 1 for head + half the list

unsuccess: 1 " " + all the list

How to control λ?

- **Rehashing:** If table is too dense / too sparse, realloc. to new table of ideal size

Designer: $\lambda_{min}, \lambda_{max}$ - allowed λ values

$\lambda_0 = \frac{\lambda_{min} + \lambda_{max}}{2}$ "ideal"

If $\lambda < \lambda_{min}$ or $\lambda > \lambda_{max}$ ...

If $\lambda < \lambda_{min}$ or $\lambda > \lambda_{max}$? Rehash!

- Alloc. new table size = n/λ_0
- Compute new hash fn h
- Copy each x,v from old to new using h
- Delete old table

Thm: Amortized time for rehashing is $1 + (2\lambda_{max}/(\lambda_{max} - \lambda_{min}))$

Token-based – See latex notes!

Hashing III

Collision Resolution: (cont.)

- Separate chaining is efficient, but uses extra space (nodes, pointers, ...)
- Can we just use the table itself?

→ **Open Addressing**

Open Addressing:

- Special entry ("empty") means this slot is unoccupied
- Assume $\lambda \leq 1$
- To insert key:

  check: $h(x)$ if not empty try
  $h(x)+i_1$
  $h(x)+i_2$
  ⋮

  $\langle i_1, i_2, i_3, \ldots \rangle$ – **Probe sequence**
- What's the best probe sequence?

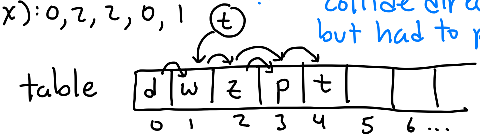
Linear Probing:

$h(x), h(x)+1, h(x)+2, \ldots$

until finding first available

Simple, but is it good?

x: d, z, p, w, t
$h(x)$: 0, 2, 2, 0, 1



t did not collide directly but had to probe 3 times!

Analysis:

Let S_{LP} = expected time for successful search

U_{LP} = " " unsuccessful "

$$\boxed{\textbf{Thm: } \quad S_{LP} = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right) \qquad U_{LP} = \frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)^2}$$

Obs: As $\lambda \to 1$ times increase rapidly

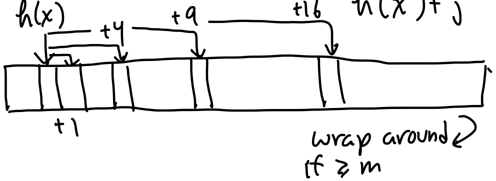
Clustering

- Clusters form when keys are hashed to nearby locations
- Spread them out!

Quadratic Probing:

$h(x), h(x)+1, h(x)+4, h(x)+9, \ldots$

$h(x)+j^2$



Analysis: Improves secondary clustering

- May fail to find empty entry
  (Try $m=4$. $j^2 \bmod 4 = 0$ or 1 but not 2 or 3)
- How bad is it? It will succeed if $\lambda < 1/2$.

Thm: If quad. probing used & m is prime, the the first $\lfloor m/2 \rfloor$ probe locations are distinct.

Pf: See latex notes.

Hashing IV

Recap:

Separate Chaining: Fastest but uses extra space (linked list)

Open Addressing:
- Linear probing:
- Quadratic probing:

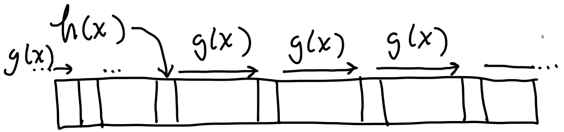
} clustering

Double Hashing:

(Best of the open-addressing methods)

- Probe sequence det'd by second hash fn. – $g(x)$

$$h(x) + \{0, g(x), 2 \cdot g(x), 3 \cdot g(x) \ldots\} \ [\text{mod } m]$$



(until finding an empty slot)

Why does bust up clusters?

Even if $h(x) = h(y)$ [collision] it is very unlikely that $g(x) = g(y)$

⇒ Probe sequences are entirely different!

Analysis: Defs:

S_{DH} = Expected search time of doub. hash. if successful

U_{DH} = Exp. if unsuccessful

Recall: Load factor $\lambda = n/m$

Thm: $S_{DH} = 1/\lambda \ln (1/1-\lambda)$

$U_{DH} = 1/(1-\lambda)$

→ Proof is nontrivial (skip)

λ:	0.5	.075	0.95	0.99
U_{DH}:	2	4	20	100
S_{DH}:	1.39	1.89	3.15	4.65

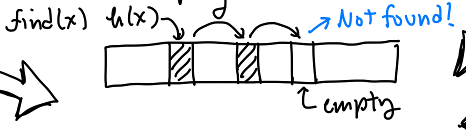
Very efficient!

Dictionary Operations:

Insert(x,v): Apply probe sequence until finding first empty slot.
- Insert (x,v) here.

(If x found along the way ⇒ duplicate key error!)

Find(x): Visit entries on probe sequence until:
- found x ⇒ return v
- hit empty ⇒ return null



Delete(x): Apply find(x)
- → Not found ⇒ error
- → Found ⇒ set to "~~empty~~" "deleted"

~~Is this right??~~

Problem: