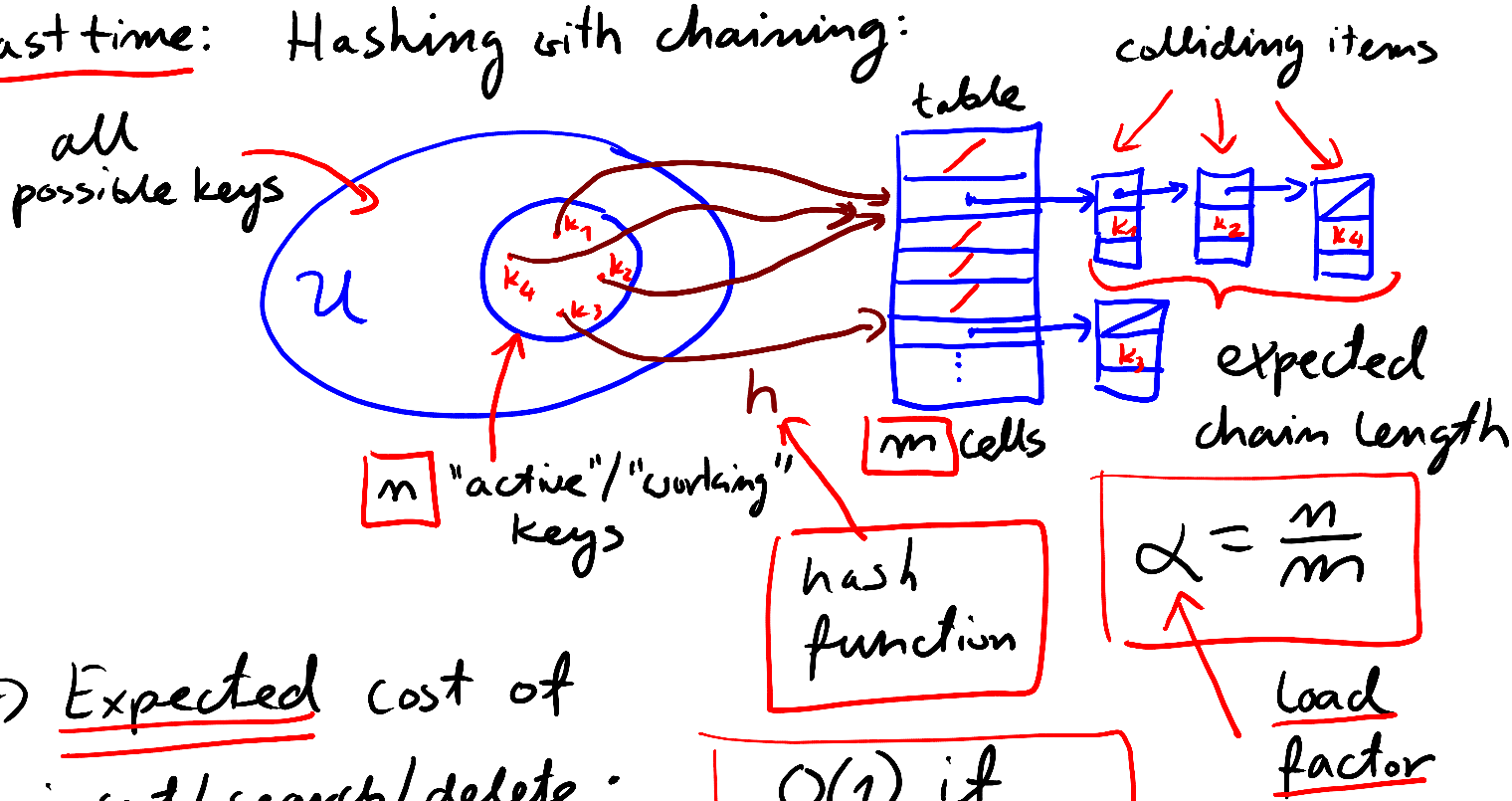
Admin:
- → PSet 2 due tonight!
- → PSet 3 out
  - → Looking at problems 3-1 to 3-4 before the quiz is a good review
  - → No Part B this time
  - → Quiz 1 next week!
    - → Review recitations on Wed
    - → NO LECTURE on Thur
    - → "Normal" recitations on Fri
    - → Practice on past quizzes
      courses.csail.mit.edu/6.006
  - → Cookies today!

See handout on Stellar

Today:
- → Table resizing
- → Amortized analysis
- → String matching & Karp-Rabin alg.
- → Rolling hash

Reading: CLRS Ch 17

Last time: Hashing with chaining:



→ Expected cost of insert/search/delete:

$$\Theta(1+\alpha)$$

$O(1)$ if $m = \Omega(n)$

Important: This assumes that:
- → h satisfies Simple Uniform Hashing Assumption (or Universal Hashing)
- → h takes $O(1)$ time to compute

"Good" hash functions:

→ Division method $h(k) = k \bmod m$

→ Multiplication method: $|\mathcal{U}| = 2^w$

$h(k) = \lfloor (a \cdot k) \bmod 2^w \rfloor >> (w-r)$ $m = 2^r$

(a: random)

Can be broken by certain sets of keys →

→ Universal hashing:

$h(k) = \lfloor (a \cdot k + b) \bmod p \rfloor \bmod m$

(a, b: random; p: large prime)

Provably good for any set of keys →

How large should the table be?

→ Want $m = \Theta(n)$ at all times

→ But: we don't know at first what n will be!

→ Our guess on m too small ⇒ slow (lots of collisions)

→ —— 11 —— m too large ⇒ wasteful on space

Key idea: Start with table being small (e.g., $m = 1$)
→ grow (& shrink) as necessary

How to grow/shrink a hash table?

Rehashing: → Choose the new size m' & new hash function h' (need a new hash fun. as size changed)

→ Build new hash table from scratch
insert each item from the old table into the new one (& discard old one when done)

⇒ Time needed:

$\Theta(n+m) = \Theta(n)$ if $m = \Theta(n)$

(This is NOT $O(1)$ time!)

How fast to grow the table?

Assume n reaches m (after an insert)

→ $m \mathrel{+}= 1$?

⇒ could need to rehash every step

⇒ n inserts cost

$$\Theta(1+2+3+\ldots+n) = \Theta(n^2) >> n!$$

(Table doubling)

→ $m \mathrel{*}= 2$? (Note: still $m = \Theta(n)$)

⇒ rehash at each 2^i-th insertion

⇒ n inserts cost

(really the next power of 2)

$$\Theta(1+2+4+8+\cdots+n) = \Theta(n)$$

(Similar in spirit to build_heap [L4])

Result: A few inserts cost $\Omega(n)$ time but $O(1)$ "on average"

Broader concept: Amortized analysis

→ Common technique in DSs. Think: Like paying rent: \$1500/month ≈ \$50/day

→ Operation has amortized cost $T(n)$ if $\forall k \geq 1$, k operations cost $\leq k \cdot T(n)$

⇒ "$T(n)$ amortized" ≈ "$T(n)$ on average"; but averaged over all ops.

Note: Some of these ops might still take long time (but most don't!)

Back to hashing:

→ Can maintain $m = \Theta(n)$ under insertions in $\Theta(1)$ amortized time

⇒ $\alpha = O(1)$ ⇒ search in $O(1)$ expected time (under SUHA/univ. hashing + $O(1)$ time eval. of h)

Delete operations?

→ Already $O(1)$ expected time as is

→ But: m can get big wrt n → wasteful!

e.g.: $n\times$ insert, $n\times$ delete

→ Solution: When n decreases to $\frac{m}{4}$ shrink to half the size (Any const. >2 works)

Exercise: What goes wrong if we use $\frac{m}{2}$?

⇒ $O(1)$ amortized cost to maintain $m = \Theta(n)$ for both inserts & deletions

But: Analysis harder → see CLRS 17.4

Sidenote: can make everything $O(1)$ time per op. in an non-amortized sense, but a bit tricky

Python: The same approach gives us resizable lists (arrays) in Python

⇒ list.append & list.pop in $O(1)$ amortized time

String matching problem: $t > s$

Given two strings $S[1 \ldots s]$ & $T[1 \ldots t]$, does S occur as a substring of T?

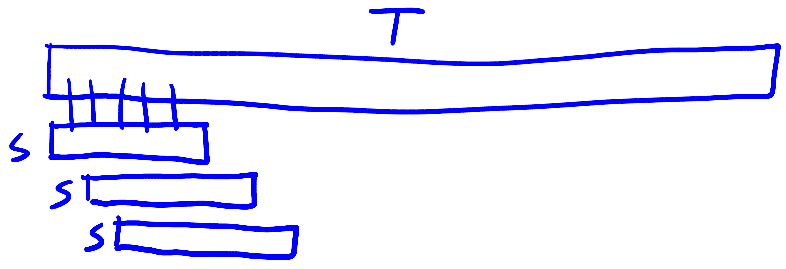
In other words: Does there exist a shift k $(0 \le k \le t-s)$

s.t. $T[k+1 \ldots k+s] = S[1 \ldots s]$? (s-character match)

Think: $S \leftarrow$ pattern $T \leftarrow$ text

E.g. S = '6.006' & T = your entire INBOX

Naive algorithm: Slide a length-s "window" through T, checking all $(t-s)$ shifts for a potential match with S



Running time? $O((t-s) \cdot s) = O(s \cdot t)$

Potentially quadratic.

# of possible shifts

checking for a match each time

Can we do better?

Key bottleneck: Need to check for a match each time

Karp-Rabin algorithm:

Idea: Quickly "pre-screen" potential matches by comparing hashes, i.e., check if $h(T[k+1 \ldots k+s]) \stackrel{?}{=} h(S[1 \ldots s])$

Note: If hashes are **different** ⇒ **no match**
but if hashes are **equal** ⇒ there still might be **NO match**

"false positives" due to collisions

⇒ If hashes match, still need to verify for match

⇒ But: probability p of "false positive" will be small for a suitable choice of h

Running time? (*)

$$O(t(1+p \cdot s)+s) = O(t) \text{ if } p \leq \frac{1}{s} !$$

linear time!

- t: computing & comparing hashes
- $p \cdot s$: dealing with false positives
- s: checking "final" match

Unfortunately: Above analysis cheats/abuses the model

Indication something is "fishy": If we take h to be a "trivial" identity hash function $h(x) = x$

Then: → prob. of false positive is 0
→ K-R alg. becomes the naïve alg.
→ BUT: (*) would still indicate $O(t)$ run time! (instead of $O(st)$)

Root of the problem: Can't really assume that computing $h(T[k+1 \ldots s])$ takes $O(1)$ time

Note: Size of $T[k+1 \ldots s]$ is s ← not a const.

⇒ Computing $h(T[k+1 \ldots s])$ should take $\Omega(s)$ time

Resulting "real" running time is

$$O\left(t\left(\underbrace{s}_{\text{"real" time needed to compute \& compare hashes}} + p \cdot s\right) + s\right) = \underbrace{O(ts)}_{\text{No improvement!}}$$

What can we do now?

→ Use a hash function that can leverage the fact that every two consecutive sliding windows differ only by two characters

Rolling hash: Provides a quick way to update hash value when we slide the window by one position:

Given $h(T[k+1 \ldots k+s])$ compute $h(T[k+2 \ldots k+s+1])$ in $O(1)$ time

How to do that?

Neat idea: Algebraization

→ Encode the window as a s-digit number (base B, for $B >$ alphabet size)

$$x = d_1 d_2 \ldots d_s \xrightarrow{h} \underbrace{\sum_{i=1}^{s} d_i B^{s-i}}_{h(x)} \pmod q$$

→ To keep sizes of #s reasonable use mod q

→ As long as q is some random prime $\approx s$

(one needs to prove that) ⇒ prob. p of false positives $\leq \frac{1}{s}$

$$y = h(x) = \sum_{i=1}^{s} d_i B^{s-i}$$

$x = d_1 \dots d_s$

Now:

→ To remove first character

$x = 0 d_2 \dots d_s$

$$y \leftarrow y - d_1 B^{s-1} \pmod q$$

→ To shift everything left

$x = d_2 \dots d_s 0$

$$y \leftarrow y \cdot B \pmod q$$

→ To specify new low-order digit l (after shift)

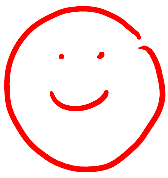
$x = d_2 \dots d_s d_{s+1}$

$$y \leftarrow y + l \pmod q$$

⇒ Updating a hash after a shift $O(1)$ time

Resulting running time of Karp-Rabin alg.:

$$O\big(s + (t-s)(1 + p \cdot s)\big) = O(t)!$$



- s: Compute the initial hash (& verify the final match)
- $(t-s)$: shifts
- 1: update the rolling hash
- $p \cdot s$: deal with false positives