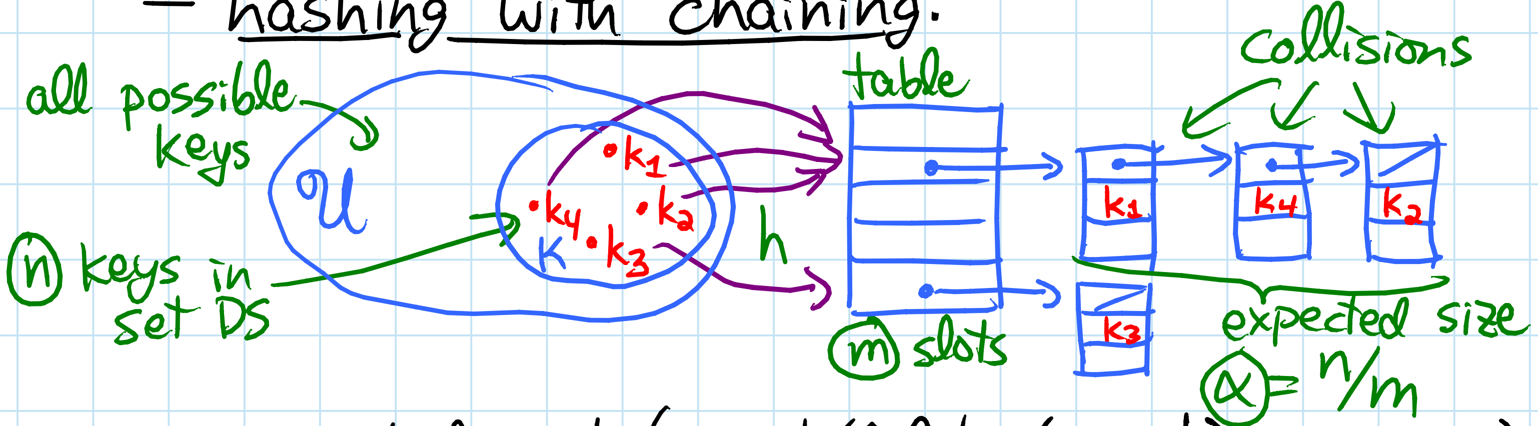
6.006 | Lecture 9 | Oct. 5. 2011

TODAY: Hashing II

- table resizing
- amortization
- string matching & Karp-Rabin
- rolling hash

Recall:

- hashing with chaining:



- expected cost (insert/delete/search): $\Theta(1+\alpha)$
    ↳ assuming simple uniform hashing
    OR universal hashing
    & hash function h takes $O(1)$ time

- division method: $h(k) = k \bmod m$
    ideally prime ↗

- multiplication method:
    $h(k) = [(a \cdot k) \bmod 2^w] \gg (w-r)$
    random ↲ ↳ w bits ↳ $m = 2^r$

How large should table be?

- want $m = \Theta(n)$ at all times
- don't know how large n will get @creation
- m too small ⇒ slow; m too big ⇒ wasteful

Idea: start small (constant)
grow (&shrink) as necessary

Rehashing: to grow or shrink table
hash function must change (m, r)
⇒ must rebuild hash table from scratch

```
for item in old table:    → for each slot:
                               for item in slot:
    insert into new table
```

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

How fast to grow? when n reaches m, say

- m += 1 ?
  ⇒ rebuild every step
  ⇒ n inserts cost $\Theta(1+2+\cdots+n) = \Theta(n^2)$
- m *= 2 ? $m = \Theta(n)$ still (r += 1)
  ⇒ rebuild at insertion 2^i
  ⇒ n inserts cost $\Theta(1+2+4+8+\cdots+n)$
  really the next power of 2 ↑
  $= \Theta(n)$
  - a few inserts cost linear time, but $\Theta(1)$ "on average"

Amortized analysis — common technique in DSs

— like paying rent: $1500/month ≈ $50/day

- operation has amortized cost $T(n)$ if k operations cost $\leq k \cdot T(n)$
- "$T(n)$ amortized" roughly means $T(n)$ "on average", but averaged over all ops.
- e.g. inserting into a hash table takes $O(1)$ amortized time

Back to hashing: maintain $m = \Theta(n) \Rightarrow \alpha = \Theta(1)$

$\Rightarrow$ support search in $O(1)$ expected time (assuming simple uniform hashing/universal)

Delete: also $O(1)$ expected as is

- space can get big with respect to n e.g. $n \times$ insert, $n \times$ delete
- solution: when n decreases to $m/4$, shrink to half the size

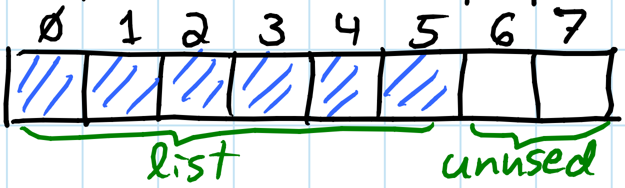
$\Rightarrow O(1)$ amortized cost for both insert & delete

- analysis harder; see CLRS 17.4

Resizable arrays:

- same trick solves Python "list" (array)

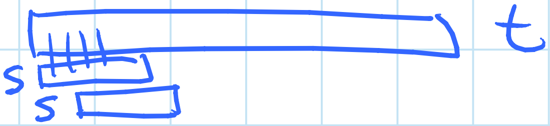
$\Rightarrow$ list.append & list.pop in $O(1)$ amortized



String matching: given two strings s & t, does s occur as a substring of t? (and if so, where & how many times?)

e.g. s = '6.006' & t = your entire INBOX

('grep' on UNIX)



Simple algorithm:

```
any(s == t[i:i+len(s)]
        for i in range(len(t)-len(s)))
```

- $O(|s|)$ time for each substring comparison

$\Rightarrow O(|s| \cdot (|t|-|s|))$ time

$= O(|s| \cdot |t|)$ potentially quadratic

Karp-Rabin algorithm:

- compare h(s) == h(t[i:i+len(s)])
- if hash values match, likely so do strings
  - can check s == t[i:i+len(s)] to be sure ~ cost $O(|s|)$
  - if yes, found match — done
  - if no, happened with probability $< \frac{1}{|s|}$

    $\Rightarrow$ expected cost is $O(1)$ per i
- need suitable hash function
- expected time $= O(|s| + |t| \cdot \text{cost}(h))$
  - naively h(x) costs $|x|$
  - we'll achieve $O(1)$!
  - idea: t[i:i+len(s)] ≈ t[i+1:i+1+len(s)]

Rolling hash ADT: maintain string x subject to

- **r()**: reasonable hash function $h(x)$
- **r.append(c)**: add letter c to end of x
- **r.skip(c)**: remove front letter from x, assuming it is c

Karp-Rabin application:

```
for c in s: rs.append(c)
for c in t[:len(s)]: rt.append(c)
if rs()==rt(): ...
for i in range(len(s), len(t)):
    rt.skip(t[i-len(s)])
    rt.append(t[i])
    if rs()==rt(): ...
```

$\}\ O(|s|)$ (first three lines)

$\}\ O(|t|)$ (loop)

$+O(\#\text{matches} \cdot |s|)$ to verify

Data structure: treat string x as a multidigit number u in base a

alphabet size ↑ e.g. 256

- $r() = u \bmod p$ for prime $p \approx |s|$ or $|t|$ (division method)

  ideally random ↗

- r stores $u \bmod p$ & $|x|$ (really $a^{|x|}$), not u

  ⇒ smaller & faster to work with ($u \bmod p$ fits in one machine word)

- r.append(c): $(u \cdot a + \text{ord}(c)) \bmod p$

  $= [(u \bmod p) \cdot a + \text{ord}(c)] \bmod p$

- r.skip(c): $[u - \text{ord}(c) \cdot (a^{|x|-1} \bmod p)] \bmod p$

  $= [(u \bmod p) - \text{ord}(c) \cdot (a^{|x|-1} \bmod p)] \bmod p$