Today: Hashing II
- table resizing
- amortization
- string matching & Karp-Rabin
- rolling hash

Recall:
- hashing with chaining:
  - all possible keys
  - \( n \) keys in set DS
  - expected cost (insert/delete/search): \( \Theta(1+\alpha) \)
  - assuming simple uniform hashing OR universal hashing
  - hash function \( h \) takes \( O(1) \) time

- division method: \( h(k) = k \mod m \)
  - ideally prime

- multiplication method:
  \[ h(k) = \left[ (a \cdot k) \mod 2^w \right] \gg (w-r) \]
  - \( a \) random \( \ll w \) 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 $\Rightarrow$ slow; $m$ too big $\Rightarrow$ wasteful

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

Rehashing: to grow or shrink table
hash function must change $(m, r)$
$\Rightarrow$ must rebuild hash table from scratch
for item in old table: $\Rightarrow$ for each slot:
insert into new table
$\Rightarrow \Theta(n+m)$ time $= \Theta(n)$ if $m = \Theta(n)$

How fast to grow? when $n$ reaches $m$, say
- $m += 1$?
  $\Rightarrow$ rebuild every step
  $\Rightarrow n$ inserts cost $\Theta(1+2+\cdots+n) = \Theta(n^2)$
- $m *= 2$? $m = \Theta(n)$ still $(r += 1)$
  $\Rightarrow$ rebuild at insertion $2^i$
  $\Rightarrow n$ 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

```
  0 1 2 3 4 5 6 7
  | | | | | | | |
list   unused
```
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 \sim cost O(|s|)
  - if yes, found match - done
  - if no, happened with probability < 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)] \approx t[i+1:i+1+len(s)]
Rolling hash ADT: maintain string \( x \) subject to
- \( r() \): reasonable hash function \( h(x) \)
- \( r.\text{append}(c) \): add letter \( c \) to end of \( x \)
- \( r.\text{skip}(c) \): remove front letter from \( x \), assuming it is \( c \)

Karp-Rabin application:
for \( c \) in \( s \):
  \( rs.\text{append}(c) \)
for \( c \) in \( t[:\text{len}(s)] \):
  \( rt.\text{append}(c) \)
  if \( rs() == rt() \): ...
for \( i \) in \( \text{range(len}(s), \text{len}(t)) \):
  \( rt.\text{skip}(t[i-\text{len}(s)]) \)
  \( rt.\text{append}(t[i]) \)
  if \( rs() == rt() \): ...

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

Data structure: treat string \( x \) as a multidigit number \( u \) in base a
  alphabet size \( \uparrow \) e.g. 256
- \( r() = u \mod p \) for prime \( p \approx |s| \) or |t|
  ideally random (division method)
- \( r \) stores \( u \mod p \) & \( |x| \) (really \( a^{|x|} \)), not \( u \)
  \( \Rightarrow \) smaller & faster to work with
  \( (u \mod p \text{ fits in one machine word}) \)
- \( r.\text{append}(c) = (u \cdot a + \text{ord}(c)) \mod p \)
  \( = [(u \mod p) \cdot a + \text{ord}(c)] \mod p \)
- \( r.\text{skip}(c) = [(u - \text{ord}(c) \cdot (a^{|x|-1} \mod p)) \mod p \]
  \( = [(u \mod p) - \text{ord}(c) \cdot (a^{|x|-1} \mod p)] \mod p \)