Lec9_Table doubling, Karp-Rabin

Size of Table

- Problems
  - want m = $\Theta(n)$ at all times
  - don't know how large n will get at creation
  - m too small ⇒ slow; m too big ⇒ wasteful
- Idea
  - Start small(constant) and grow (or 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
  - m += 1?
    ⇒ rebuild every step
    ⇒ n inserts cost $\Theta(1+2+3+ \ldots + n) = \Theta(n^2)$
  - m *=2? m = $\Theta(n)$ still (r += 1)
    ⇒ rebuild at insertion 2^i
    ⇒ n insert cost $\Theta(1+2+4+8+ \ldots + n)$
    ⇒ $\Theta(n)$

Amortized Analysis

- Amortized Analysis
  - Operation has amortized cost T(n) if k operations cost <= k* 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 = $\Theta(n)$ ⇒ $\alpha = \Theta(n)/n = \Theta(1)$ ⇒ support search in O(1) expected time
- Delete
  - Also O(1) expected as is
    - Space can get big with respect to n e.g. n X insert, n X delete
  - Solution : when n decrease to m/4 shrink to half the size ⇒ O(1) amortized cost for both insert and delete → analysis is h
- Resizable Array :
  - same trick solves Python "list" (array)

    - ⇒ list.append and list.pop in O(1) amortized

String Matching

Given two strings s and t, does s occur as a substring of t? (and if so, where and how many times?)

- Simple Algorithm

```
any(s==t[i:i+len(s)] for i in range(len(t) - len(s))
    → O(|s|) time for each substring comparison
    ⇒ O(|s|*(|t|-|s|)) time
    = O(|s|*|t|) : potential quadratic
```

- Karp-Rabin
    - 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 < 1/|s|
          ⇒ expected cost is O(1) per i
    - need suitable hash function
    - expected time = O(|s| + |t| * cost(h))
        - Naively h(x) costs |x|
        - We'll achieve O(1)
        - idae : 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) on string x
- r.append(c) : add letter c to end of string x
- r.skip(c) : remove front letter from string 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-len(s)])
    if rs() == rt()
        check whether s == t[i-len(s)+1 : i+1]
        if equal : found match
        else : happens with probability <= 1/|s|
```

- Running time : O(|s|+|t| + #match*|s|)

Data Structure

Treat string x as a multi digit number in base a where a denotes the alphabet size, e.g, 256

- r() = u mod p for prime p~=|s| or |t| (division method)
- r stores u mod p and |x| (really a^|x|) not u
  ⇒ smaller and faster to work with (u mod p fits in one machine word)
- r.append(c)

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

- r.skip(c)

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

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