

Lecture 9: Hashing II

Lecture Overview

- Table Resizing
- Amortization
- String Matching and Karp-Rabin
- Rolling Hash

Recall:

Hashing with Chaining:

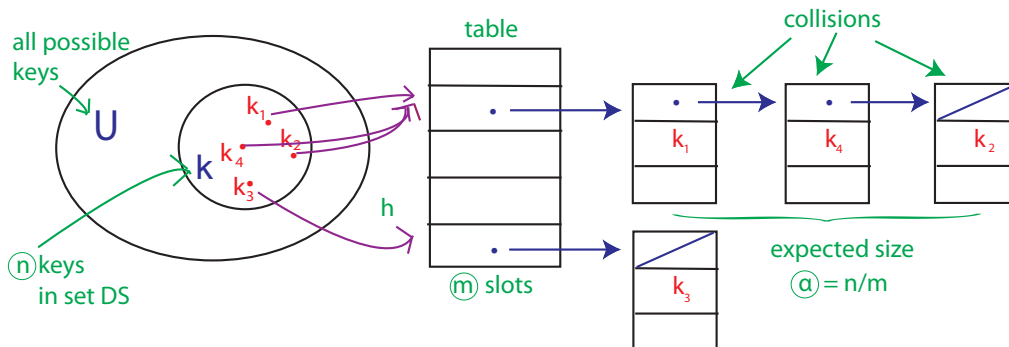


Figure 1: Hashing with Chaining

Expected cost (insert/delete/search): $\Theta(1 + \alpha)$, assuming simple uniform hashing *OR* universal hashing & hash function h takes $O(1)$ time.

Division Method:

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

where m is ideally prime

Multiplication Method:

$$h(k) = [(a \cdot k) \bmod 2^w] \gg (w - r)$$

where a is a random odd integer between 2^{w-1} and 2^w , k is given by w bits, and $m = \text{table size} = 2^r$.

How Large should Table be?

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

Idea:

Start small (constant) and grow (or shrink) as necessary.

Rehashing:

To grow or shrink table hash function must change (m, r)

- \implies must rebuild hash table from scratch
 - for item in old table: \rightarrow for each slot, for item in slot
 - insert into new table
- $\implies \Theta(n + m)$ time = $\Theta(n)$ if $m = \Theta(n)$

How fast to grow?

When n reaches m , say

- $m + = 1$?
 - \implies rebuild every step
 - $\implies n$ inserts cost $\Theta(1 + 2 + \dots + n) = \Theta(n^2)$
- $m * = 2$? $m = \Theta(n)$ still ($r + = 1$)
 - \implies rebuild at insertion 2^i
 - $\implies n$ inserts cost $\Theta(1 + 2 + 4 + 8 + \dots + n)$ where n is really the next power of 2 = $\Theta(n)$
- a few inserts cost linear time, but $\Theta(1)$ “on average”.

Amortized Analysis

This is a common technique in data structures — like paying rent: \$1500/month \approx \$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 $m = \Theta(n) \implies \alpha = \Theta(1) \implies$ support search in $O(1)$ expected time (assuming simple uniform or universal hashing)

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 $\implies O(1)$ amortized cost for both insert and delete — analysis is harder; see CLRS 17.4.

Resizable Arrays:

- same trick solves Python “list” (array)
- \implies list.append and list.pop in $O(1)$ amortized

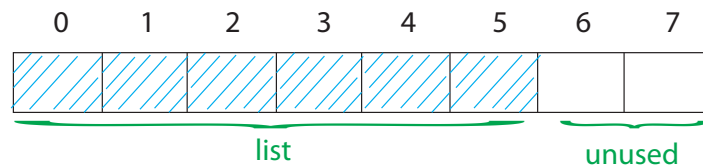


Figure 2: Resizable Arrays

String Matching

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

E.g. $s = \text{'6.006'}$ and $t =$ your entire INBOX (‘grep’ on UNIX)

Simple Algorithm:

any($s == t[i : i + \text{len}(s)]$ for i in range($\text{len}(t) - \text{len}(s)$))
 — $O(|s|)$ time for each substring comparison
 $\implies O(|s|) \cdot (|t| - |s|)$ time
 $= O(|s| \cdot |t|)$ potentially quadratic

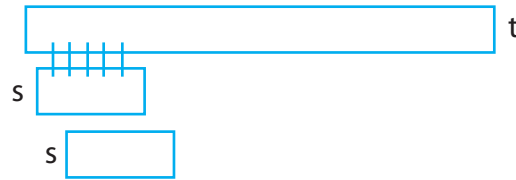


Figure 3: Illustration of Simple Algorithm for the String Matching Problem

Karp-Rabin Algorithm:

- Compare $h(s) == h(t[i : i + \text{len}(s)])$
- If hash values match, likely so do strings
 - can check $s == t[i : i + \text{len}(s)]$ to be sure $\sim \text{cost } O(|s|)$
 - 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: $t[i : i + \text{len}(s)] \approx t[i + 1 : i + 1 + \text{len}(s)]$.

Rolling Hash ADT

Maintain string x subject to

- $r()$: reasonable hash function $h(x)$ on string x
- $r.\text{append}(c)$: add letter c to end of string x
- $r.\text{skip}(c)$: 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(): ...
```

This first block of code is $O(|s|)$

```

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

```

The second block of code is $O(|t|) + O(\# \text{ matches} - |s|)$ to verify.

Data Structure:

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

- $r() = u \bmod p$ for (ideally random) prime $p \approx |s|$ or $|t|$ (division method)
- r stores $u \bmod p$ and $|x|$ (really $a^{|x|}$), not u
 \implies smaller and faster to work with ($u \bmod p$ fits in one machine word)
- $r.append(c)$: $(u \cdot a + \text{ord}(c)) \bmod p = [(u \bmod p) \cdot a + \text{ord}(c)] \bmod p$
- $r.skip(c)$: $[u - \text{ord}(c) \cdot (a^{|u|-1} \bmod p)] \bmod p$
 $= [(u \bmod p) - \text{ord}(c) \cdot (a^{|x|-1} \bmod p)] \bmod p$

MIT OpenCourseWare
<http://ocw.mit.edu>

6.006 Introduction to Algorithms
Fall 2011

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.