6.006	Lecture 10	Oct. 13, 2016

TODAY: Introduction to Advanced Hashing
- indicator random variables, careful analysis
- universal hashing
- open addressing
- perfect hashing
- cuckoo hashing

Recall the basics (a little more formally):

Prehash function maps keys to integers in $\mathcal{U} = \{0, 1, \ldots, u-1\}$ "universe"

Hash function $h: \mathcal{U} \to \{0, 1, \ldots, m-1\}$ (m = table size)

load factor $\alpha = n/m$

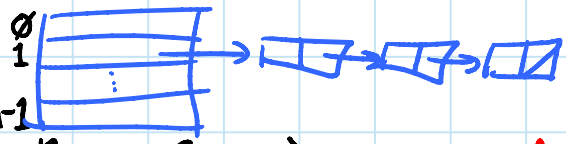
Hash table stores n distinct keys $x_1, x_2, \ldots, x_n$

Simple Uniform Hashing Assumption: (SUHA)

$\Pr\{h(x_i) = j\} = 1/m$ for all $1 \le i \le n$ & $0 \le j < m$

& $h(x_i)$ is independent from all $h(x_{\neq i})$

Hashing with chaining:



- # keys in slot/list $j = \sum_{i=1}^{n} I\{h(x_i) = j\}$

$= \begin{cases} 1 & \text{if } h(x_i) = j \\ 0 & \text{else} \end{cases}$

indicator random variable

↓ linearity of expectation

- $E[\#\text{keys in slot/list } j] = \sum_{i=1}^{n} \Pr\{h(x_i) = j\}$

$= n/m$ assuming SUHA

$\Rightarrow E[\text{running time of Insert/Delete/Search}] = O(1 + \underbrace{n/m}_{\alpha})$

What is random? using Pr, E, etc...

- keys $x_1, x_2, \ldots, x_n$? "average case" – avoid
- works, but less interesting

➡ hash function h – better!

- don't need to assume anything about input!

~ but:

SUHA is bad: implies h is chosen uniformly at random among all hash functions

- consists of $\Theta(u)$ integers $\in \{0, 1, \ldots, m-1\}$

$\Rightarrow \Theta(u)$ space to store & $\Theta(u)$ time to generate! (random # calls)

Universal family $\mathcal{H}$ of hash functions satisfies

$$\Pr_{h \in \mathcal{H}} \{h(x) = h(y)\} \leq 1/m \quad \text{for all } x \neq y \in \mathcal{U}$$

- e.g. $h_{ab}(x) = [(a \cdot x + b) \bmod p] \bmod m$ [L8]

  random $\in \mathcal{U}$ (1 to 1); fixed prime $\geq u$

  $\mathcal{H} = \{h_{ab} \mid a, b \in \mathcal{U}\}$

- just 2 random values to store/pick: $a, b \in \mathcal{U}$

$\Rightarrow L_i$ = # keys hashing to same slot as x_i

$$= \sum_{i'=1}^{n} \mathbb{I}_{h \in \mathcal{H}} \{h(x_{i'}) = h(x_i)\}$$

$$= 1 + \sum_{i' \neq i} \mathbb{I}_{h \in \mathcal{H}} \{h(x_{i'}) = h(x_i)\}$$

$\Rightarrow \mathbb{E}_{h \in \mathcal{H}}[L_i] = 1 + \sum_{i' \neq i} \Pr_{h \in \mathcal{H}} \{h(x_{i'}) = h(x_i)\}$ ↓ linearity of expectation

$$\leq 1 + \sum_{i' \neq i} 1/m = 1 + \frac{n-1}{m} < 1 + \alpha$$

Open addressing: no linked lists

- store all n items in array of size $m \geq n$
- in fact, maintain $m \geq 2 \cdot n$ (or any const. factor > 1)
- define probe sequence $h(x,0), h(x,1), h(x,2), \ldots$ for where to put each key x
  - e.g. linear probing: $h(x,i) = (x+i) \bmod m$
  - great cache performance
  - parking lot problem: big car clusters likely grow
  - fancier methods better, but only for $m \approx n$
  - e.g. changing Python dict to linear probing $\Rightarrow$ 20% faster
- insert(x): (assuming $x \notin T$)

```
for i = 0, 1, 2, ...
    if T[h(x,i)] empty:
        T[h(x,i)] = x
        return
```

[Cohen, Georgiou, Razavi 6.851 final project, 2012]

linear probing + 5-wise independence (e.g. $a+bx+cx^2+dx^3+ex^4$) $\Rightarrow$ $O(1)$ expected

[Pagh, Pagh, Ružić - STOC 2007]

- search(x):

```
for i = 0, 1, 2, ...
    if T[h(x,i)] = x: return YES
    if T[h(x,i)] empty: return NO
```

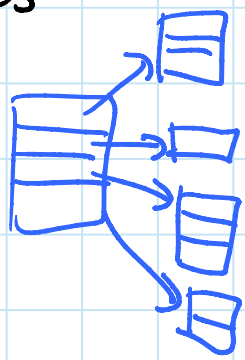
- delete(x): (assuming $x \in T$)

```
for i = 0, 1, 2, ...
    if T[h(x,i)] = x:
        T[h(x,i)] = GHOST   ← can't use empty!
        return
```

- if table gets 10% ghosts (say), rebuild
  - amortized cost remains $O(1)$/operation

Perfect hashing: [Fredman, Komlós, Szemeredi – J.ACM 1984]

- guarantee search takes exactly 2 probes
- primary hash with $m = n$
- instead of chains, resolve collisions with another hash table!
  - for k keys colliding in a slot, use table of size $\Theta(k^2)$
  - no collisions by Birthday Paradox (with prob. $\geq 1/2$)
- space is (surprisingly) $O(n)$ in expectation (wow!)
- with effort, can support insert/delete too
  ~ but used most often when set of keys is static



Cuckoo hashing: [Pagh & Rodler – J.Algorithms 2004]

- guarantee search takes 1 or 2 probes (& can be parallelized!)
- 2 tables of size $m \geq 2n$ (or any const. factor > 1)
- 2 hash functions, h_1 & h_2 for tables T_1 & T_2 resp.
- search(x): check $T_1[h_1(x)]$ & $T_2[h_2(x)]$
- insert(x):

```
if T1[h1(x)] empty: T1[h1(x)] = x
elif T2[h2(x)] empty: T2[h2(x)] = x
else: flip(y = T1[h1(x)]) → T2
        ↳ if T2[h2(y)] occupied:
            recursively flip(T2[h2(y)]) → T1
        ↳ T2[h2(y)] = y
      T1[h1(x)] = x
↳ if get into infinite loop (or spend > lg n): rehash
```

- $O(1)$ amortized for $\Theta(\lg n)$-wise independence