6.006 Lecture 7: Hashing 3

- Rolling hashes
- Open Addressing
- Uniform hashing + analysis
- Advanced topics: universal, perfect, & cryptographic hashing

- CLRS 11.4, 11.3.3, 41.5

Open Addressing

- No linked lists
- Collision? Store elsewhere in hash table
- More collisions? Probe more; may need to probe m-1 times to find an empty slot

- Hash function of key \( k \) is now a sequence of probes \( \langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle \)
  
  sequence must be a permutation of 0, 1, 2, \ldots, m-1
  
  a key maps to a permutation

- Clearly, load factor \( \alpha \leq 1 \).
Rabin-Karp String matching & Rolling hashes

Given pattern $P[1..m]$ \{ lists of characters \}

text $T[1..n]$ 

does $P$ occur in $T$?

E.g. find "BFG" in "AATCGGC..."

Idea:
- Compute $h(P)$
- For each length-$m$ window of $T$, $T[i..i+m-1]$
  
  \[
  \begin{array}{c}
  AATCGGC \\
  \underline{ATCGC} \\
  \end{array}
  \]

  Compute $h(T[i..i+m-1])$ and compare to $h(P)$
  if $=$: check to see if really a match
  if $\neq$: move on to next $i$

- Use a hash function $h$ s.t. can compute $h(T[i..i+m-1])$ from $h(T[i-1..i+m])$ easily
  $\Rightarrow$ rolling hash
Example of a rolling hash

Alphabet \( A, C, G, T \)

\[ h = \text{value of string } \% p \text{ for prime } p \]

\[ T = \text{TATTACGTT} \]
\[ 1 3 0 3 3 0 1 2 3 4 \] < base 4
\[ 1 8 4 0 9 1 1 0 \]
\[ y 3 \mod p = 1009 \]

\( \downarrow \) delete \( C \)

\[ T = \text{TATTACGTT} \]
\( \uparrow \) add \( G \)

value of leading \( C = 1 \cdot 4^8 = 1.960 \)

so dropping leading \( C \Rightarrow h = 3453 - 690 = 502 \mod p \)

\( \uparrow \text{shifting and adding a } G: \)

\[ h = 4 \cdot 502 + 2 = 1001 \mod p \]

\[ \text{shift } \]

\[ \text{add } \]

\[ \text{mulitply } \times 1 \]

\[ \text{cost of moving } h \text{ to next window} \]
**Insert** \((k, v)\):

for \(i\) in range \((m)\):
    if \(T[h(k,i)] == \text{None}\): \(T[h(k,i)] = (k, v)\)
    return
raise Exception('full')

**Search** \((k)\):

for \(i\) in range \((m)\)
    if \(T[h(k,i)] == \text{None}\): return None  # not in table
    if \(T[h(k,i)][0] == k\): return \(T[h(k,i)]\)
return None

**Delete** \((k)\):

# tricky! setting \(T[h(k,i)] = \text{None}\) may cause
search to fail (e.g. deleting the first
element inserted in the
example causes the third
not to be found

* Find key
* Replace by 'Deleted'
* Skip over 'Deleted' in
  search but use 'Deleted' slots in insert.
How to construct \( h(k,i) \)

- What do we want?
  - For chaining, we want simple uniform hashing:
    each key is equally likely to hash to any slot.
  - For open addressing, we want uniform hashing:
    each key is equally likely to hash to any of the \( m! \) probe sequences (permutations of 0, 1, \ldots, m-1).
  - Harder to achieve, but double hashing works well.

**Linear probing**

- Start with an ordinary hash function \( h'(k) \)
- \( h(k,i) = (h'(k)+i) \mod m \)
- Start at \( h'(k) \) and scan sequentially
- Not good: only \( m \) possible sequences, leads to clustering

**Double hashing**

- \( h(k,i) = (h_2(k)+i \cdot h_2(k)) \mod m \)
- to ensure \( h(k,*) \) hits all slots, make \( h_2(k) \) and \( m \) relatively prime. Ex: \( m=2^r, h_2(k) \) odd
**Open addressing vs Chaining**

- Cost explodes as \( n \) approaches 1.
- No memory allocation (except to resize), cache efficient.
- Hard to find a really good \( h \).
- Easier to implement in hardware.

**Chaining**

- Cost rises gently with \( n \).
- Allocates memory as chains grow (constant overhead).
- Easy to find a good \( h \).

**Advanced topics in hashing**

- **Universal hashing** (back to chaining)
  
  For any \( h \) there are collisions; if we are unlucky all the keys may hash to 1 slot.

  Solution: Don't use a fixed \( h \); choose it at random.

  Universal hashing: random selection of \( h \) should guarantee
  
  \[
  \Pr[k_1, k_2 \text{ collide}] \leq \frac{1}{m}
  \]

  \[
  h(k) = ((ak + b) \mod p) \mod m \quad \text{(last lecture) works.}
  \]

  random

- **Perfect hashing**: \( \Theta(1) \) worst-case search for fixed set of keys.
  
  - Primary table stores pointers to secondary tables + their hash function.
  - Make secondaries large enough so prob. of any collision \( \leq \frac{1}{2} \).
  - Try secondary hashes until no collisions at all.

  Space is still \( \Theta(n) \) (expected).