Lecture 6: Hashing II: Table Doubling, Rolling Hash, Karp-Rabin

Lecture Overview

• Table Resizing

• Amortization

• DNA Comparison, Karp-Rabin Rolling Hash

Readings

CLRS Chapter 17 and 32.2.

Recall:

• Hashing with Chaining:

![Diagram of Chaining in a Hash Table]

Figure 1: Chaining in a Hash Table
• **Simple Uniform Hashing Assumption** *(Silently used in all of this lecture)*

Each key is equally likely to be hashed to any slot of table, independent of where other keys are hashed.

\[
\Rightarrow \text{average } \# \text{ keys per slot is } \alpha = \frac{n}{m}
\]

\[
\Rightarrow \text{expected time to search, insert, delete } = O(1 + \alpha).
\]

**Caution:** The above bound assumes that the application of the hash function \( h(\cdot) \) takes \( O(1) \) time. Sometimes this is not the case, e.g. if the keys are strings. Then the keys need to be processed into numbers and then hashed to \( \{0, 1, \ldots, m-1\} \). Then the above bound is scaled by the time needed for applying \( h \).

• **Good Hash Functions:**
  
  – **Division Method:** \( h(k) = k \mod m \)
    
    *Good Practice:* \( m \) is a prime number & not close to a power of 2 or 10.
  
  – **Multiplication Method:** \( h(k) = [a \cdot k \mod 2^w] \gg (w - r) \), where
    
    - \( \gg \) denotes the “shift right” operator,
    - \( 2^r \) is the table size (= \( m \)),
    - \( w \) the bit-length of the machine words,
    - and \( a \) is chosen to be an odd integer between \( 2^{(w-1)} \) and \( 2^w \).
    
    *Good Practice:* \( a \) not too close to \( 2^{(w−1)} \) or \( 2^w \).

\[
\begin{align*}
\text{w} & \quad \text{k} \\
\text{x} & \quad \text{a} \\
\text{ignore} & \quad \text{keep} & \quad \text{ignore} \\
\text{r} & \quad \text{w-r}
\end{align*}
\]

\[\equiv \]

\[+\]

\[\text{product as sum lots of mixing}\]

Figure 2: Multiplication Method
How Large should Table be?

- want \( m = \Theta(n) \) at all times

- **Why?**
  \( m \) too small \( \implies \) slow (recall operations take expected time \( \Theta(1 + \alpha) \), where \( \alpha = \frac{n}{m} \));
  \( m \) too big \( \implies \) wasteful

**Challenge:** Don’t know how large \( n \) will get at creation.

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

**Table Resizing with Rehashing:**

To change \( m \) build new hash table from scratch:

Allocate table of size \( m \);

For each item in old 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 \( \Theta(1 + 2 + \cdots + n) = \Theta(n^2) \)

- \( m \ast = 2? \) \( m = \Theta(n) \) still
  \( \implies \) rebuild at insertion \( 2^i \), pay \( 2^{i+1} \) (see Figure 3)
  \( \implies n \) inserts cost \( \Theta(1 + 2 + 4 + 8 + \cdots + n) \) 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 \( \approx \$ 50/day \)

- if a sequence of \( n \) operations has total cost \( \leq n \cdot T(n) \), then each operation has \emph{amortized cost} \( T(n) \)

- “\( T(n) \) amortized” roughly means \( T(n) \) “on average”, but averaged over all ops.

- e.g. inserting into a hash table (with doubling) takes \( O(1) \) amortized time.
Back to Hashing:

Maintain \( m = \Theta(n) \) so also support search in \( O(1) \) expected time assuming simple uniform hashing.

Delete:

Also \( O(1) \) expected time

- space can get big with respect to \( n \) e.g. \( n \times \) insert, \( n \times \) delete

- solution: when \( n \) decreases to \( m/4 \), shrink table to \( m/2 \)
  \[ \Rightarrow \] \( O(1) \) amortized cost for both insert and delete —
  —analysis is trickier; (see CLRS 17.4).
Rolling Hash: Human vs Chimp

Given two strings $S$ and $T$, find the longest common substring of two strings.
Naive algorithm: $\Theta(n^4)$.
Naive + binary search: $\Theta(n^3 \log n)$.
Winner algorithm from last lecture runs in time $\Theta(n^2 \log n)$, using hash tables:

For all possible lengths $\ell$:

- (Step 1) Insert all substrings $s$ of $S$ of length $\ell$ into a hash table using some hash function $h$;
- (Step 2) For all substrings $t$ of $T$ of length $\ell$, check if position $h(t)$ of the dictionary is occupied. If yes (say by substring $s$ of string $S$), compare $s$ and $t$. If strings agree return $s$ and $t$ and exit.

Analysis:

Outer Loop: using binary search on the length of the longest common substring, only $O(\log n)$ iterations are needed!

For every $\ell$:

Step 1:

- there are $n - \ell + 1$ substrings $s$ of $S$ of length $\ell$;
- need to convert each of them into an integer; how?
  think of $s := S[i : i + \ell]$ as a multi-digit number base $b$, where $b$ is larger than the alphabet size
  
  $s \mapsto S[i] \cdot b^{\ell-1} + S[i + 1] \cdot b^{\ell-2} + \ldots + S[i + \ell - 1]$

- use hash function $h(s) = s \mod m$
- How to compute $h(s)$ without writing down the above expression? Mod arithmetic magic!

Claim: For all integers $a, b$:

\begin{align*}
    a + b \mod m &= ((a \mod m) + (b \mod m)) \mod m \\
    a \cdot b \mod m &= ((a \mod m) \cdot (b \mod m)) \mod m
\end{align*}

- hence, computation of hash takes time $O(\ell)$ for each substring $s$.
  $\implies$ total time for Step 1: $O(\ell \cdot (n - \ell + 1)) = O(n^2)$.  

\[5\]
Step 2:

- there are \( n - \ell + 1 \) substrings \( t \) of \( T \) of length \( \ell \);
- every hash operation takes \( O(\ell) \), plus another potential \( O(\ell) \) for comparing strings if hashes match.
- \( \Rightarrow \) total time for Step 1: \( O(\ell \cdot (n - \ell + 1)) = O(n^2) \).

Overall time is \( O(n^2 \log n) \).

Our goal: Drop time down to \( O(n^2) \) initialization time + \( O(n \log n) \) execution time. By making both Step 1 and Step 2 take \( O(n) \).

Idea: use \( h_1 := h(S[i : i+\ell]) \) to compute \( h_2 := h(S[i+1 : i+\ell+1]) \) in \( O(1) \) time.

How?

Go from

\[
h_1 = S[i] \cdot b^{\ell-1} + S[i+1] \cdot b^{\ell-2} + \ldots + S[i+\ell-1] \mod m
\]

to

\[
h_2 = S[i+1] \cdot b^{\ell-1} + S[i+2] \cdot b^{\ell-2} + \ldots + S[i+\ell] \mod m.
\]

Without recomputing \( h_2 \) from scratch.

Magic Again — Called Rolling Hash, and introduced by Karp-Rabin:

\[
h_2 = \left( S[i+1] \cdot b^{\ell-1} + S[i+2] \cdot b^{\ell-2} + \ldots + S[i+\ell] \right) \mod m =
\]

\[
= \left( S[i] \cdot b^{\ell-1} + S[i+1] \cdot b^{\ell-2} + \ldots + S[i+\ell-1] \right) b + S[i+\ell] - S[i] \cdot b \mod m
\]

\[
= \left( h_1 \cdot b + S[i+\ell] - S[i] \cdot (b^\ell \mod m) \right) \mod m.
\]

Now Step 1 takes time \( O(n) \) overall. What about Step 2? Also, time \( O(n) \), except if there are too many spurious matches of hash values, which do not actually result in matching substrings: Each comparison takes \( O(\ell) \) and if many unsuccessful comparisons are made this could increase the time for Step 2 to \( O(n\ell) \).

But: For each substring of \( T \) the probability of a false-positive hash-value match is \( < \frac{n}{m} \) (under the simple uniform hashing assumption). Hence, choosing \( m = O(n^2) \), the expected time to process a substring \( t \) that does not match a substring of \( S \) is \( < \frac{1}{n} \cdot O(\ell) = O(1) \) \((1/n \) is the probability that the hash value of \( t \) collides with the hash value of a substring of \( S \) and \( O(\ell) \) is the time to carry out the string comparison).

Hence, \( O(n^2) \) time to allocate memory for a table of size \( m = O(n^2) \) + \( O(n \log n) \) execution time.
NOTE: We can get away with a hash table of size $\Theta(n)$, rather than $\Theta(n^2)$, while avoiding a large comparison cost from spurious hash-value matches with a further trick: if a substring $s$ occupies position $h(s)$ of the hash table, we store at that position both $s$ and a signature of $s$, produced by taking the multi-digit number corresponding to the string mod $p$, where $p$ is a prime number larger than $\ell$ (and different than $m$). Now if position $h(t)$ of the hash table contains a substring $s$ then we first compare the signatures of $s$ and $t$, and only if these agree (probability $< 1/p$ for spurious matches under the simple uniform hashing assumption for the signature hash function), we compare the strings $s$ and $t$. In this case, the expected time to process a spurious match is $< 1/p \cdot O(\ell) = O(1)$, even if the hash table has size $O(n)$. To avoid incurring $O(\ell)$ cost for each signature computation, we do rolling hashing for the signatures as well :-). In this computation, we implicitly assumed that the values produced by the hash function $h$ and the signature function are independent, and that both functions satisfy the simple uniform hashing assumption.