Lecture 5: Hashing I: Chaining, Hash Functions

Lecture Overview

- Dictionaries
- Motivation — fast DNA comparison
- Hash functions
- Collisions, Chaining
- Simple uniform hashing
- “Good” hash functions

Readings

CLRS Chapter 11. 1, 11. 2, 11. 3.

Dictionary Problem

Dictionary: Abstract Data Type (ADT) maintaining a set of items, each with a key.

E.g. (phonebook) keys are names, and their corresponding items are phone numbers
E.g.2 (real dictionary) keys are english words, and their corresponding items are dictionary-entries

Operations to Support:

- insert(item): add item to set
- delete(item): remove item from set
- search(key): return item with key if it exists

Assumption: items have distinct keys (or that inserting new one clobbers old)

- Balanced BSTs solve in $O(\log n)$ time per operation (in addition to inexact searches like nextlargest). What is the $O(\cdot)$ notation hiding? Reality: $O(\log n) \cdot \text{key\_length}$ — important distinction if key is not a number or key-length is larger than machine word.

- Our goal: $O(1)$ time per operation (again we mean $O(1) \cdot \text{key\_length}$). Using an idea called ‘Rolling Hash’ in the next lecture, we will sometimes manage to avoid paying the key\_length multiplicative penalty (on average).
Motivation

**Example Application:** How close is chimp DNA to human DNA? Find the longest common substring of two strings, e.g. ALGORITHM vs. ARITHMETIC.

**Naive algorithm?**

INPUT: two strings S1, S2 of length $n$.

for $l = n$, $n-1$, ..., 1
  for all substrings $x_1$ of $S_1$ of length $l$
    for all substrings $x_2$ of $S_2$ of length $l$
      if $x_1 == x_2$ return $l$;

i.e. compare all possible substrings of the two DNA sequences — needs $\Theta(n^4)$ operations.

**Improvements?** Can do binary search (how?) on the length of the longest common substring, dropping down the number of operations to $\Theta(n^3 \log n)$.

→ Using dictionaries can drop this down to $\Theta(n^2 \log n)$. Here is how:

For all possible lengths $l$:

- Insert all substrings of $S_1$ of length $l$ into a dictionary;
  (there are $O(n)$ such substrings, and each insertion takes $O(1) \cdot 1$ time)

- for all $O(n)$ substrings of $S_2$ of length $l$ do a $O(1) \cdot 1$ look-up!

Running time is $O(n^3)$. Now replacing the outer loop with Binary Search reduces this to $O(n^2 \log n)$. 
How do we solve the dictionary problem?

A simple approach would be a direct access table. This means items would need to be stored in an array, indexed by key.

![Direct-access table](image)

**Figure 1:** Direct-access table

**Problems:**

1. keys must be nonnegative integers (or using two arrays, integers)
2. large key range \(\Rightarrow\) large space e.g. one key of \(2^{256}\) is bad news.

**2 Solutions:**

**Solution 1:** map key space to integers “Everything is number.” - Pythagoras.

- In Python: hash (object) where object is a number, string, tuple, etc. or object implementing `__hash__`
- Misnomer: should be called “prehash”
- Ideally, \(x = y \iff\) hash\( (x) = \) hash\( (y)\)
- Python applies some heuristics e.g. hash\( (‘\\$B’) = 64 = \) hash\( (‘\\$\$C’)\)
- Object’s key should not change while in table (else cannot find it anymore)

**Solution 2:** hashing (verb from ‘hache’ = hatchet, Germanic)

- Reduce universe \(U\) of all keys (say, integers) down to reasonable size \(m\) for table
- idea: \(m \approx n\), where \(n = |K|\), \(K =\) set of keys in dictionary
• hash function \( h: \mathcal{U} \rightarrow \{\emptyset, 1, \ldots, m-1\} \)

• think of \( m \) as a number that fits in a machine word
  (if 32 bits, then \( m \) can be up to about a billion, so dictionary can be quite large; if
  that is not enough can use two words, etc.)

\[ U \] : universe of all possible keys

\[ K \] : actual keys

\[ h(k_1) \]

\[ h(k_3) \]

\[ h(k_2) = h(k_4) \] (collision)

\[ \emptyset \]

\[ 1 \]

\[ \text{item1} \]

\[ \text{item3} \]

\[ \text{problem} \]

\[ h(k_2) = h(k_4) \] (collision)

\[ m-1 \]

Figure 2: Mapping keys to a table

• two keys \( k_i, k_j \in K \) collide if \( h(k_i) = h(k_j) \)

*How do we deal with collisions?*

There are two ways

1. Chaining: TODAY

2. Open addressing: NEXT LECTURE
Chaining

Linked list of colliding elements in each slot of table

- Search must go through whole list $T[h(key)]$
- Worst case: all keys in $k$ hash to same slot $\Rightarrow \Theta(n)$ per operation

Simple Uniform Hashing: an Assumption:

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

- let $n =$ number of keys stored in table, $m =$ number of slots in table
- average $\sharp$ keys per slot $= n/m =: \alpha$ — the load factor
  - Why? Throw $n$ balls into $m$ bins uniformly at random. Average $\#$ balls/bin is $\frac{n}{m}$.

Expected performance of chaining: assuming simple uniform hashing

- Expected time to search $= O(1 + \alpha)$
  - pay 1 to apply hash function and access slot; then pay $\alpha$ to search the list.
- Expected time to insert/delete $= O(1 + \alpha)$

$\Rightarrow$ the performance is $O(1)$ if $\alpha = O(1)$ i.e. $m = \Omega(n)$.
Two Concrete Hash Functions

**Division Method:** $h(k) = k \mod m$

- $k_1$ and $k_2$ collide when $k_1 \equiv k_2 \pmod{m}$, i.e. when $m$ divides $|k_1 - k_2|$
- fine if keys you store are uniform random (probability of collision=1/$m$)
- but if keys are $x, 2x, 3x, \ldots$ (regularity) and $x \& m$ have common divisor $d$ then use only $1/d$-th of the table. **Because** $i \cdot x \equiv (i + \frac{m}{d}) \cdot x \pmod{m}$. (This is likely if $m$ has a small divisor, e.g. 2)
- if $m = 2^r$ then only look at $r$ bits of key!
- **Good Practice:** $m$ is a prime number & not close to a power of 2 or 10 (to avoid common regularities in keys)
- **BUT:** Inconvenient to find a prime number; division slow.

**Multiplication Method:** [Look at figure first]
$h(k) = [(a \cdot k) \mod 2^w] \gg (w - r)$, 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 $2^w$.

**Key Lesson:** Multiplication and bit extraction are faster than division.

![Figure 4: Multiplication Method](image-url)