Hashing

Lecture 15

March 15, 2011

Part I

Hash Tables

Dictionary Data Structure

- A universe $\mathcal{U}$ of keys that have a total order: numbers, strings, etc.
- Data structure to store a subset $\mathbf{S} \subseteq \mathcal{U}$
- **Operations:**
  - **Search**/lookup: given $\mathbf{x} \in \mathcal{U}$ is $\mathbf{x} \in \mathbf{S}$?
  - **Insert**: given $\mathbf{x} \notin \mathbf{S}$ add $\mathbf{x}$ to $\mathbf{S}$.
  - **Delete**: given $\mathbf{x} \in \mathbf{S}$ delete $\mathbf{x}$ from $\mathbf{S}$
- **Static** structure: $\mathbf{S}$ given in advance or changes very infrequently, main operations are lookups.
- **Dynamic** structure: $\mathbf{S}$ changes rapidly so inserts and deletes as important as lookups.

Dictionary Data Structures

Common solutions:

- Static:
  - Store $\mathbf{S}$ as a *sorted* array
  - Lookup: binary search in $\mathbf{O}(\log |\mathbf{S}|)$ time (comparisons)
- Dynamic:
  - Store $\mathbf{S}$ in a *balanced* binary search tree
  - Lookup, Insert, Delete in $\mathbf{O}(\log |\mathbf{S}|)$ time (comparisons)

Dictionary Data Structures

Question: "Should Tables be Sorted?"
(also title of famous paper by Turing award winner Andy Yao)

Hashing is a widely used & powerful technique for dictionaries.

Motivation:

- Universe $\mathcal{U}$ may not be (naturally) totally ordered
- Keys correspond to large objects (images, graphs etc) for which comparisons are very expensive
- Want to improve "average" performance of lookups to $O(1)$ even at cost of extra space or errors with small probability: many applications for fast lookups in networking, security, etc.

Hashing and Hash Tables

Hash Table data structure:

- A (hash) table/array T of size m (the table size)
- A hash function $h : \mathcal{U} \to \{0, \ldots, m-1\}$
- Item $x \in \mathcal{U}$ hashes to slot $h(x)$ in T

Given $S \subseteq \mathcal{U}$. How do we store S and how do we do lookups?

Ideal situation:

- Each element $x \in S$ hashes to a distinct slot in T. Store x in slot $h(x)$
- Lookup: given $y \in \mathcal{U}$ check if $T[h(y)] = y$. $O(1)$ time!

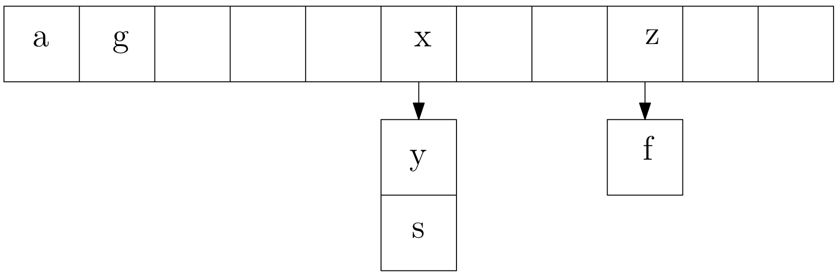
Collisions unavoidable. Several different techniques to handle them.

Handling Collisions: Chaining

Collision: $\mathbf{h(x) = h(y)}$ for some $\mathbf{x} \neq \mathbf{y}$.

Chaining to handle collisions:

- For each slot $\mathbf{i}$ store all items hashed to slot $\mathbf{i}$ in a linked list. $\mathbf{T[i]}$ points to the linked list
- Lookup: to find if $\mathbf{y} \in \mathcal{U}$ is in $\mathbf{T}$, check the linked list at $\mathbf{T[h(y)]}$. Time proportion to size of linked list.



Handling Collisions

Several other techniques:

- Open addressing
- ...
- Cuckoo hashing

Understanding Hashing

Does hashing give $\mathbf{O(1)}$ time per operation for dictionaries?

Questions:

- Complexity of evaluating $\mathbf{h}$ on a given element?
- Relative sizes of the universe $\mathcal{U}$ and the set to be stored $\mathbf{S}$.
- Size of table relative to size of $\mathbf{S}$.
- Worst-case vs average-case vs randomized (expected) time?
- How do we choose $\mathbf{h}$?

Understanding Hashing

- Complexity of evaluating $\mathbf{h}$ on a given element? Should be small.
- Relative sizes of the universe $\mathcal{U}$ and the set to be stored $\mathbf{S}$: typically $|\mathcal{U}| \gg |\mathbf{S}|$.
- Size of table relative to size of $\mathbf{S}$. The **load factor** of $\mathbf{T}$ is the ratio $\mathbf{n/t}$ where $\mathbf{n} = |\mathbf{S}|$ and $\mathbf{m} = |\mathbf{T}|$. Typically $\mathbf{n/t}$ is a small constant smaller than $\mathbf{1}$.
  Also known as the **fill factor**.

Main and interrelated questions:

- Worst-case vs average-case vs randomized (expected) time?
- How do we choose $\mathbf{h}$?

Single hash function

- $\mathcal{U}$: universe (very large).
- Assume $\mathbf{N} = |\mathcal{U}| \gg \mathbf{m}$ where $\mathbf{m}$ is size of table $\mathbf{T}$. In particular assume $\mathbf{N} \geq \mathbf{m}^2$ (very conservative).
- Fix hash function $\mathbf{h} : \mathcal{U} \to \{0, \ldots, \mathbf{m} - 1\}$.
- $\mathbf{N}$ items hashed to $\mathbf{m}$ slots. By pigeon hole principle there is some $\mathbf{i} \in \{0, \ldots, \mathbf{m} - 1\}$ such that $\mathbf{N}/\mathbf{m} \geq \mathbf{m}$ elements of $\mathcal{U}$ get hashed to $\mathbf{i}$!
- Implies that there is a set $\mathbf{S} \subseteq \mathcal{U}$ where $|\mathbf{S}| = \mathbf{m}$ such that all of $\mathbf{S}$ hashes to same slot!

Lesson: For every hash function there is a very bad set!

Picking a hash function

- Hash function are often chosen in an ad hoc fashion. Implicit assumption is that input behaves well.
- Theory and sound practice suggests that a hash function should be chosen properly with guarantees on its behavior.

Parameters: $\mathbf{N} = |\mathcal{U}|$, $\mathbf{m} = |\mathbf{T}|$, $\mathbf{n} = |\mathbf{S}|$

- $\mathcal{H}$ is a *family* of hash functions: each function $\mathbf{h} \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $\mathbf{h}(\mathbf{x})$)
- $\mathbf{h}$ is chosen *randomly* from $\mathcal{H}$ (typically uniformly at random). Implicitly assumes that $\mathcal{H}$ allows an efficient sampling.
- Randomized guarantee: should have the property that for any *fixed* set $\mathbf{S} \subseteq \mathcal{U}$ of size $\mathbf{m}$ the expected number of collisions for a function chosen from $\mathcal{H}$ should be "small". Here the expectation is over the randomness in choice of $\mathbf{h}$.

Picking a hash function

Question: Why not let $\mathcal{H}$ be the set of *all* functions from $\mathcal{U}$ to $\{0, 1, \ldots, m-1\}$?

- Too many functions! A random function has high complexity!

Question: Are there good and compact families $\mathcal{H}$?

- Yes... But what it means for $\mathcal{H}$ to be good and compact.

Uniform hashing

Question: What are good properties of $\mathcal{H}$ in distributing data?

- Consider any element $x \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then x should go into a random slot in T. In other words $\Pr[h(x) = i] = 1/m$ for every $0 \leq i < m$.
- Consider any two distinct elements $x, y \in \mathcal{U}$. Then if $h \in \mathcal{H}$ is picked randomly then the probability of a collision between x and y should be at most $1/m$. In other words $\Pr[h(x) = h(y)] = 1/m$ (cannot be smaller).
- Second property is stronger than the first and the crucial issue.

Definition

A family hash function $\mathcal{H}$ is **2-universal** if for all distinct $x, y \in \mathcal{U}$, $\Pr[h(x) = h(y)] = 1/m$ where m is the table size.

Note: The set of all hash functions satisfies stronger properties!

Analyzing Uniform Hashing

- $\mathbf{T}$ is hash table of size $\mathbf{m}$.
- $\mathbf{S} \subseteq \mathcal{U}$ is a *fixed* set of size $\mathbf{m}$.
- $\mathbf{h}$ is chosen randomly from a uniform hash family $\mathcal{H}$.
- $\mathbf{x}$ is a *fixed* element of $\mathcal{U}$. Assume for simplicity that $\mathbf{x} \notin \mathbf{S}$.

Question: What is the *expected* time to look up $\mathbf{x}$ in $\mathbf{T}$ using $\mathbf{h}$ assuming chaining used to resolve collisions?

Analyzing Uniform Hashing

Question: What is the *expected* time to look up $\mathbf{x}$ in $\mathbf{T}$ using $\mathbf{h}$ assuming chaining used to resolve collisions?

- The time to look up $\mathbf{x}$ is the size of the list at $\mathbf{T[h(x)]}$: same as the number of elements in $\mathbf{S}$ that collide with $\mathbf{x}$ under $\mathbf{h}$.
- Let $\ell(\mathbf{x})$ be this number. We want $\mathrm{E}[\ell(\mathbf{x})]$
- For $\mathbf{y} \in \mathbf{S}$ let $\mathbf{A_y}$ be the even that $\mathbf{x}, \mathbf{y}$ collide and $\mathbf{D_y}$ be the corresponding indicator variable.

Analyzing Uniform Hashing

Continued...

$$\ell(x) = \sum_{y \in S} D_y$$

$$\begin{aligned}
\Rightarrow \mathrm{E}[\ell(x)] &= \sum_{y \in S} \mathrm{E}[D_y] \quad \text{linearity of expectation} \\
&= \sum_{y \in S} \Pr[h(x) = h(y)] \\
&= \sum_{y \in S} \frac{1}{m} \quad \text{since } \mathcal{H} \text{ is a uniform hash family} \\
&= |S|/m \leq 1
\end{aligned}$$

Analyzing Uniform Hashing

Question: What is the *expected* time to look up x in T using h assuming chaining used to resolve collisions?

Answer: $O(1)$!

Comments:

- $O(1)$ expected time also holds for insertion.
- Analysis assumes static set S but holds as long as S is a set formed with at most $O(m)$ insertions and deletions.
- *Worst-case* look up time can be large! How large? $\Omega(\log n / \log \log n)$.

Rehashing, amortization and...

... making the hash table dynamic

Previous analysis assumed fixed $\mathbf{S}$ of size $\simeq \mathbf{m}$.

Question: What happens as items are inserted and deleted?

- If $|\mathbf{S}|$ grows to more than $\mathbf{cm}$ for some constant $\mathbf{c}$ then hast table performance clearly degrades
- If $|\mathbf{S}|$ stays around $\simeq \mathbf{m}$ but incurs many insertions and deletions then the initial random hash function is no longer random enough!

Solution: Rebuild hash table periodically!

- Choose a new table size based on current number of elements in table.
- Choose a new random hash function and rehash the elements.
- Discard old table and hash function.

Question: When to rebuild? How expensive?

Rebuilding the hash table

- Start with table size $\mathbf{m}$ where $\mathbf{m}$ is some estimate of $|\mathbf{S}|$ (can be some large constant).
- If $|\mathbf{S}|$ grows to more than twice current table size, build new hash table (choose a new random hash function) with double the current number of elements. Can also use similar trick if table size falls below quarter the size.
- If $|\mathbf{S}|$ stays roughly the same but more than $\mathbf{c}|\mathbf{S}|$ operations on table for some chosen constant $\mathbf{c}$ (say $\mathbf{10}$), rebuild.

Amortize cost of rebuilding to previously performed operations. Rebuilding ensures $\mathbf{O(1)}$ expected analysis holds even when $\mathbf{S}$ changes. Hence $\mathbf{O(1)}$ expected look up/insert/delete time *dynamic* data dictionary data structure!

Some math required...

Lemma

Let $\mathbf{p}$ be a prime number,
$\mathbf{x}$: an integer number in $\{1, \ldots, \mathbf{p}-1\}$.
$\implies$ *There exists a unique $\mathbf{y}$ s.t. $\mathbf{xy} = 1 \mod \mathbf{p}$.*

In other words: For every element there is a unique inverse.
$\implies$ $\mathbb{Z}_{\mathbf{p}} = \{0, 1, \ldots, \mathbf{p}-1\}$ when working module $\mathbf{p}$ is a field.

Proof of lemma

Claim

Let $\mathbf{p}$ be a prime number. For any $\alpha, \beta, \mathbf{i} \in \{1, \ldots, \mathbf{p}-1\}$ s.t. $\alpha \neq \beta$, we have that $\alpha\mathbf{i} \neq \beta\mathbf{i} \mod \mathbf{p}$.

Proof.

Assume for the sake of contradiction $\alpha\mathbf{i} = \beta\mathbf{i} \mod \mathbf{p}$. Then

$$
\begin{aligned}
&\mathbf{i}(\alpha - \beta) = 0 \mod \mathbf{p} \\
\implies&\quad \mathbf{p} \text{ divides } \mathbf{i}(\alpha - \beta) \\
\implies&\quad \mathbf{p} \text{ divides } \alpha - \beta \\
\implies&\quad \alpha - \beta = 0 \\
\implies&\quad \alpha = \beta.
\end{aligned}
$$

And that is a contradiction. $\square$

Proof of lemma...

Uniqueness.

Proof.

Follows immediately from the above claim. Indeed, assume the claim is false and there are two distinct numbers $\mathbf{y}, \mathbf{z} \in \{1, \ldots, \mathbf{p}-1\}$ such that

$$\mathbf{xy} = 1 \mod \mathbf{p} \quad \text{and} \quad \mathbf{xz} = 1 \mod \mathbf{p}.$$

But this contradicts the above claim (set $\mathbf{i} = \mathbf{x}$, $\alpha = \mathbf{y}$ and $\beta = \mathbf{z}$). □

Proof of lemma...

Existence

Proof.

By claim, for any $\alpha \in \{1, \ldots, \mathbf{p}-1\}$ we have that $\{\alpha * 1 \mod \mathbf{p}, \alpha * 2 \mod \mathbf{p}, \ldots, \alpha * (\mathbf{p}-1) \mod \mathbf{p}\} = \{1, 2, \ldots, \mathbf{p}-1\}$.
$\implies$ There exists a number $\mathbf{y} \in \{1, \ldots, \mathbf{p}-1\}$ such that $\alpha\mathbf{y} = 1 \mod \mathbf{p}$. □

Constructing Universal Hash Families

Parameters: $\mathbf{N = |\mathcal{U}|}$, $\mathbf{m = |T|}$, $\mathbf{n = |S|}$

- Choose a **prime** number $\mathbf{p \geq N}$. $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ is a field.
- For $\mathbf{a, b} \in \mathbb{Z}_p$, $\mathbf{a \neq 0}$, define the hash function $\mathbf{h_{a,b}}$ as $\mathbf{h_{a,b}(x) = ((ax + b) \mod p) \mod m}$.
- Let $\mathcal{H} = \{h_{a,b} \mid a, b \in \mathbb{Z}_p, a \neq 0$. Note that $|\mathcal{H}| = p(p-1)$.

Theorem

$\mathcal{H}$ is a $\mathbf{2}$-universal hash family.

Comments:

- Hash family is of small size, easy to sample from.
- Easy to store a hash function ($\mathbf{a, b}$ have to be stored) and evaluate it.

Constructing Universal Hash Families

Theorem

$\mathcal{H}$ is a (2)-universal hash family.

Proof.

Fix $\mathbf{x, y} \in \mathcal{U}$. What is the probability they will collide if $\mathbf{h}$ is picked randomly from $\mathcal{H}$?

- Let $\mathbf{a, b}$ be bad for $\mathbf{x, y}$ if $\mathbf{h_{a,b}(x) = h_{a,b}(y)}$
- **Claim:** Number of bad pairs is at most $\mathbf{p(p-1)/m}$.
- Total number of hash functions is $\mathbf{p(p-1)}$ and hence probability of a collision is $\leq \mathbf{1/m}$. □

Some Lemmas

Lemma

If $\mathbf{x} \neq \mathbf{y}$ then for any $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_\mathbf{p}$ such that $\mathbf{a} \neq \mathbf{0}$, $\mathbf{ax} + \mathbf{b} \mod \mathbf{p} \neq \mathbf{ay} + \mathbf{b} \mod \mathbf{p}$.

Proof.

If $\mathbf{ax} + \mathbf{b} \mod \mathbf{p} = \mathbf{ay} + \mathbf{b} \mod \mathbf{p}$ then $\mathbf{a}(\mathbf{x} - \mathbf{y}) \mod \mathbf{p} = \mathbf{0}$ and $\mathbf{a} \neq \mathbf{0}$ and $(\mathbf{x} - \mathbf{y}) \neq \mathbf{0}$. However, $\mathbf{a}$ and $(\mathbf{x} - \mathbf{y})$ cannot divide $\mathbf{p}$ since $\mathbf{p}$ is prime and $\mathbf{a} < \mathbf{p}$ and $(\mathbf{x} - \mathbf{y}) < \mathbf{p}$. $\square$

Some Lemmas

Lemma

If $\mathbf{x} \neq \mathbf{y}$ then for each $(\mathbf{r}, \mathbf{s})$ such that $\mathbf{r} \neq \mathbf{s}$ and $\mathbf{0} \leq \mathbf{r}, \mathbf{s} \leq \mathbf{p} - \mathbf{1}$ there is exactly one $\mathbf{a}, \mathbf{b}$ such that $\mathbf{ax} + \mathbf{b} \mod \mathbf{p} = \mathbf{r}$ and $\mathbf{ay} + \mathbf{b} \mod \mathbf{p} = \mathbf{s}$.

Proof.

Solve the two equations:

$$\mathbf{ax} + \mathbf{b} = \mathbf{r} \mod \mathbf{p} \text{ and } \mathbf{ay} + \mathbf{b} = \mathbf{s} \mod \mathbf{p}$$

We get $\mathbf{a} = \frac{\mathbf{r} - \mathbf{s}}{\mathbf{x} - \mathbf{y}} \mod \mathbf{p}$ and $\mathbf{b} = \mathbf{r} - \mathbf{ax} \mod \mathbf{p}$. $\square$

Proof of Claim

Proof.

Let $\mathbf{a}, \mathbf{b} \in \mathbb{Z}_p$ such that $\mathbf{a} \neq \mathbf{0}$ and $\mathbf{h_{a,b}(x) = h_{a,b}(y)}$.

- Let $\mathbf{ax + b \mod p = r}$ and $\mathbf{ay + b \mod \; = s \mod p}$.
- Collision if and only if $\mathbf{r = s \mod m}$.
- Number of pairs $(\mathbf{r}, \mathbf{s})$ such that $\mathbf{r} \neq \mathbf{s}$ and $\mathbf{0} \leq \mathbf{r}, \mathbf{s} \leq \mathbf{p} - \mathbf{1}$ and $\mathbf{r = s \mod m}$ is $\mathbf{p(p-1)/m}$.
- From previous lemma for each bad pair $(\mathbf{a}, \mathbf{b})$ there is a unique pair $(\mathbf{r}, \mathbf{s})$ such that $\mathbf{r = s \mod m}$. Hence total number of bad pairs is $\mathbf{p(p-1)/m}$.
- Prob of $\mathbf{x}$ and $\mathbf{y}$ to collide:

$$\frac{\#\text{ bad pairs}}{\#\text{pairs}} = \frac{\mathbf{p(p-1)/m}}{\mathbf{p(p-1)}} = \frac{\mathbf{1}}{\mathbf{m}}.$$

□

Perfect Hashing

Question: Can we make look up time $\mathbf{O(1)}$ in worst case?

Yes for static dictionaries but then space usage is $\mathbf{O(m)}$ only in expectation.

Take away points

- Hashing is a powerful and important technique for dictionaries. Many practical applications.
- Randomization fundamental to understanding hashing.
- Good and efficient hashing possible in theory and practice with proper definitions (universal, perfect, etc).
- Related ideas of creating a compact fingerprint/sketch for objects is very powerful in theory and practice.
- Many applications in practice.