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 $S \subseteq \mathcal{U}$

**Operations:**
- **Search**/lookup: given $x \in \mathcal{U}$ is $x \in S$?
- **Insert**: given $x \not\in S$ add $x$ to $S$.
- **Delete**: given $x \in S$ delete $x$ from $S$

**Static** structure: $S$ given in advance or changes very infrequently, main operations are lookups.

**Dynamic** structure: $S$ changes rapidly so inserts and deletes as important as lookups.

Dictionary Data Structures

Common solutions:

- **Static**:
  - Store $S$ as a *sorted array*
  - Lookup: binary search in $O(\log |S|)$ time (comparisons)

- **Dynamic**:
  - Store $S$ in a *balanced* binary search tree
  - Lookup, Insert, Delete in $O(\log |S|)$ 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 $h : \mathcal{U} \rightarrow \{0, \ldots, m - 1\}$
- 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:** \( h(x) = h(y) \) for some \( x \neq y \).

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

```
   a  g  x  z
   ^   v
   y  s  f
```

Several other techniques:
- Open addressing
- . . .
- Cuckoo hashing
Does hashing give \(O(1)\) time per operation for dictionaries?

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

Complexity of evaluating \(h\) on a given element? Should be small.

Relative sizes of the universe \(U\) and the set to be stored \(S\): typically \(|U| \gg |S|\).

Size of table relative to size of \(S\). The **load factor** of \(T\) is the ratio \(n/t\) where \(n = |S|\) and \(m = |T|\). Typically \(n/t\) is a small constant smaller than 1. Also known as the **fill factor**.

Main and interrelated questions:
- Worst-case vs average-case vs randomized (expected) time?
- How do we choose \(h\)?
Single hash function

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

- $\mathcal{H}$ is a family of hash functions: each function $h \in \mathcal{H}$ should be efficient to evaluate (that is, to compute $h(x)$)
- $h$ 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 $S \subseteq \mathcal{U}$ of size $m$ 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 $h$. 

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

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

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

The time to look up \( x \) is the size of the list at \( T[h(x)] \): same as the number of elements in \( S \) that collide with \( x \) under \( h \).

Let \( \ell(x) \) be this number. We want \( E[\ell(x)] \).

For \( y \in S \) let \( A_y \) be the even that \( x, y \) collide and \( D_y \) be the corresponding indicator variable.
\[ \ell(x) = \sum_{y \in S} D_y \]

\[ \Rightarrow E[\ell(x)] = \sum_{y \in S} E[D_y] \text{ linearity of expectation} \]

\[ = \sum_{y \in S} \Pr[h(x) = h(y)] \]

\[ = \sum_{y \in S} \frac{1}{m} \text{ since } \mathcal{H} \text{ is a uniform hash family} \]

\[ = |S|/m \leq 1 \]

**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 $S$ of size $\simeq m$.

Question: What happens as items are inserted and deleted?

- If $|S|$ grows to more than $cm$ for some constant $c$ then hash table performance clearly degrades
- If $|S|$ stays around $\simeq m$ 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 $m$ where $m$ is some estimate of $|S|$ (can be some large constant).
- If $|S|$ 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 $|S|$ stays roughly the same but more than $c|S|$ operations on table for some chosen constant $c$ (say 10), rebuild.

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

**Lemma**

Let $p$ be a prime number, $x$ an integer number in $\{1, \ldots, p-1\}$. 

$\implies$ There exists a unique $y$ s.t. $xy = 1 \mod p$.

In other words: For every element there is a unique inverse.

$\implies \mathbb{Z}_p = \{0, 1, \ldots, p-1\}$ when working module $p$ is a field.

**Proof of lemma**

**Claim**

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

**Proof.**

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

\[
i(\alpha - \beta) = 0 \mod p\]

$\implies$ $p$ divides $i(\alpha - \beta)$

$\implies$ $p$ divides $\alpha - \beta$

$\implies$ $\alpha - \beta = 0$

$\implies$ $\alpha = \beta$.

And that is a contradiction.
Proof of lemma...

Uniqueness.

Proof.

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

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

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

Existence

Proof.

By claim, for any $\alpha \in \{1, \ldots, p - 1\}$ we have that

$$\{\alpha \times 1 \mod p, \alpha \times 2 \mod p, \ldots, \alpha \times (p - 1) \mod p\} = \{1, 2, \ldots, p - 1\}.$$

$\implies$ There exists a number $y \in \{1, \ldots, p - 1\}$ such that $\alpha y = 1 \mod p$. 

□
Constructing Universal Hash Families

Parameters: \( N = |U|, m = |T|, n = |S| \)

- Choose a prime number \( p \geq N \). \( \mathbb{Z}_p = \{0, 1, \ldots, p - 1\} \) is a field.
- For \( a, b \in \mathbb{Z}_p, a \neq 0 \), define the hash function \( h_{a,b} \) as
  \[ h_{a,b}(x) = ((ax + b) \mod p) \mod m. \]
- 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 2-universal hash family.

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

---

Constructing Universal Hash Families

**Theorem**

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

**Proof.**

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

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

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

### Proof.

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

---

### Lemma

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

### Proof.

Solve the two equations:

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

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

---
Proof of Claim

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

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

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

Perfect Hashing

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

Yes for static dictionaries but then space usage is \( O(m) \) 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.