Hashing

April 1, 2003

1 Hash Table

Hash table - a data-structure for storing a set of items
Fast access - constant time.
$S$ - set of $n$ elements.
$U$ - the “universe” (i.e., $S \subseteq U$)
$x$ - elements in the set can belong to a huge set.
Thus, using an array $A[x]$ might be extremely inefficient.
Idea: map the elements of the universe $U$ into a much smaller range.
Find a mapping $h$ that maps $x \in S$ to an integer $h(x)$ between 1 and $n$,
s.t. for any $x \neq y$ we have $h(x) \neq h(y)$
If so, we can just store $x$ in $T[h(x)]$.
Problem: slow, complicated.

$$h : U \rightarrow \{0, 1, \ldots, m - 1\}$$

Ultimately, $m$ should be as close to $n$ as possible.

1.1 Finding a good hashing function.

What we want:
1. Simple.
2. Fast to compute.
3. “independent of $S$”.
4. Maps different elements of $S$ to different slots.

Impossible: once $h$ is fixed, we can always find a bad set $S$. (i.e., Adversary knows the hash function.)
Idea: Use a random hash function.

1.2 Handling Collusions

Two elements $x, y \in S$ collide, if $h(x) = h(y)$. 
1.2.1 Chaining

Store all the elements that fall in the same bucket in a single linked list.

\[
\begin{array}{cccccc}
G & H & | & R & O & A & L \\
\downarrow & & & \downarrow & & & \\
M & & I & & T & & S
\end{array}
\]

\(l(x)\) - length of the list \(T[h(x)]\)
Time to insert/search for \(x\) is \(O(1 + l(x))\).

1.3 Random Hash Function

Simple uniform hashing function assumption:
if \(x \neq y\) then \(\Pr h(x) = h(y) = 1/m\).
\(C_{x,y}\) - indicator variable that is 1 if \(x\) and \(y\) collide.

\[l(x) = \sum_{y \in S} C_{x,y}\]
Thus, the expected insertion/find time for \(x\) is: \(\mathbb{E}[1 + l(x)] = 1 + \sum_{y \in S} \mathbb{E}[C_{x,y}]\)
But \(\mathbb{E}[C_{x,y}] = 0 \ast ? + 1 \ast \Pr h(x) = h(y) = 1/m\).
Thus, the expected time to insert/search for \(x\) is:

\[1 + \sum_{y \in S} \mathbb{E}[C_{x,y}] = 1 + \frac{n}{m}.
\]

Definition 1.1 The load of a hash table, denoted by \(\alpha\) is the ratio \(n/m\) (number of elements stored in the hash table divided by the total size of the hash table).

Lemma 1.2 Using random hash function, the expected access time is \(O(1 + \alpha)\) where \(\alpha\) is the load of the hash table.

We can do weird things and maintain the list of a bucket as a balanced binary tree.
This would give \(O(1 + \log \alpha)\) for insert/search.

One can maintain the set of elements in a bucket using a hash table. This sounds silly, but in fact, can be made to work quite nicely. See next lecture.

2 Open Addressing

Another option, is to use a regular array for the hash table, and everytime we have a collision, to look in a different place.

\((h_0, \ldots, h_{m-1})\) - sequence of hashing functions.
We probe for an element in \(h_0(x)\) if this fails, we probe at \(h_1(x)\). And so on.
Assumption: For any \(x\) the sequence \((h_0(x), \ldots, h_{m-1}(x))\) is a permutation of \(0, \ldots, m-1\).
**OpenAddrSearch**

```
for i ← 0 to m – 1
    if T[h_i(x)] = x
        return h_i(x)
    if T[h_i(x)] = empty
        return “absent”
return “full”
```

Insertion is similar.

**Definition 2.1** Strong Uniform Hashing Assumption: For any \( x \),

\[
(h_0(x), \ldots, h_{m-1}(x))
\]

is a random permutation.

What is the expected search time under this assumption?

\[
\Pr h_0(x) \text{ not empty} = \frac{n}{m}.
\]

If we had a collision, the algorithm continues, on a table excluding the position \( \beta = h_0(x) \) (which we “removed”). Furthermore, we know that there are \( n – 1 \) elements. Thus, the expected search time is

\[
T(n, m) = 1 + \frac{n}{m} T(n – 1, m – 1)
\]

To solve this recurrence, we need some intuition. Note that \( T(n – 1, m – 1) \) is the expected search time in a table with a load \( \frac{n-1}{m-1} \) of size \( m-1 \). \( T(n, m) \) is the expected search time in a table with \( n \) elements and \( m \) elements, having load \( \frac{n}{m} \).

However, \( \frac{n-1}{m-1} < \frac{n}{m} \). Namely, \( T(n, m) \) is the search time in a table which is bigger, and has higher load. Thus, \( T(n – 1, m – 1) \leq T(n, m) \). Thus,

\[
T(n, m) \leq 1 + \frac{n}{m} T(n, m).
\]

\[
(1 – \frac{n}{m}) T(n, m) \leq 1
\]

\[
T(n, m) \leq \frac{m}{m-n}.
\]

**Lemma 2.2** Under the strong uniform hashing assumption, searching in a hash table with load \( \alpha = n/m \) takes \( O\left(\frac{1}{1-\alpha}\right) \) expected time.

Note, that our random probe assumption is too strong. There are several alternatives in practice:

1. **Linear probing:** \( h_i(x) = (h(x) + i) \mod m \).
   
   Bad: long runs of collisions.

2. **Quadratic probing:** \( h_i(x) = (h(x) + i^2) \mod m \)
   
   Better, but still if \( h(x) = h(y) \) they would share the same sequence of probes.

3. **Double hashing:** \( h_i(x) = (h(x) + i \cdot h'(x)) \mod m \).
   
   Works reasonably well in practice.
2.0.1 Deleting from open-address hash table

Delete $x$: Just find $x$ in the hash-table and delete it.
   Namely, follow the probe sequence for $h_0(x), h_1(x), \ldots$ and remove $x$ when you encounter it.

What is the problem? Imagine we insert $x$ and then $y$ and assume that they collide. If we delete $x$, and then search for $y$ we will not find it.

Solution: Mark entries in the hash table that we deleted from as “wasted”. Note that we can insert into a wasted bucket.

How to keep the running time down? After a long sequence of operations, all entries would be wasted.

Solution:???
   Idea: Rebuild the hash table if half of the entries are wasted.
   What to do, if we insert to the hash table, and the load approaches 1 (i.e., the table is almost full)?
   Rule: If number elements in the table exceeds $m/4$ then double the table size (i.e., $2m$), and rehash everything in it.
   Question: Running time?

Definition 2.3 Amortized running time: The average running time per operation.

Lemma 2.4 Under the strong universal hashing assumption, one can perform insertion/deletion in constant amortized time per operation.

Proof: Whenever we insert an element, we put $1$ (1 dollar = 1 time unit) in the bank for doubling the hash table. The hash-table is being rebuild only after $m/4$ operations since the last rebuild. Namely, we have in the bank $m/4$ dollars, which can pay for the $O(m)$ time to rebuild the table. Thus, insertion take amortized constant time.
   Similar argument works for deletion.

2.1 Universal Hashing

Here we describe how to fullfill the weak uniform hashing assumption.

Definition 2.5 A set $H$ of hash functions is universal if for any items $x \neq y$ we have

$$\Pr h(x) = h(y) = \frac{1}{m}.$$  

For the simplicity of exposition, we will consider $U = 0, \ldots, m-1 \times 0, \ldots, m-1$, and $m$ is a prime.

Namely, an element in our universe is a pair of numbers $(u, v)$.

For a pair of numbers, $0 \leq a, b \leq m$, let

$$h_{a,b}(x, y) = (ax + by) \mod m,$$
and

\[ H = \{ h_{a,b} \mid 0 \leq a, b < m \}. \]

**Lemma 2.6** \( H \) is universal.

**Proof:** Consider two distinct elements \((x, x') \neq (y, y')\). We have a collision if

\[ h_{a,b}(x, x') = h_{a,b}(y, y') \]

\[ (ax + bx') \mod m = (ay + by') \mod m \]

\[ a(x - y) \equiv b(y' - x) \mod m \]

Assume that \( x \neq y \), and observe that since \( m \) is a prime, \( x - y \) has a unique inverse element \( mod m \) (which we denote as \( 1/(x - y) \)). Thus, the above is equivalent to

\[ a \equiv \frac{b(y' - x)}{x - y} \mod m. \]

Since we pick \( a \) and \( b \) randomly, the probability for this to happen is \( 1/m \). Namely, the probability for collision is \( 1/m \).

So, we can now generate a universal hash function quickly. Thus, whenever we build a hash table, we randomly choose a random hash function (by randomly picking \( a \) and \( b \)), and use \( h_{a,b} \) as our hash function.

In the next lecture, we will see even better solution...