Hashing

Dictionary ADT. Supports

Membership

Insertion

Deletion

in expected $O(1)$ time.
But not predecessor.

Suppose string size $n$ set of values in $[0..m-1]$.

$$h: U \rightarrow [0..m-1]$$

where $h$ is an easy-to-compute function.

$A$ is an array of buckets.

Insert $(x)$: store $x$ in $A[h(x)]$

Delete $(x)$: remove $x$ from $A[h(x)]$

Member $(x)$: look for $x$ in $A[h(x)]$.

Want $h$ to scatter the items in $S$ so that the average non-empty list has length $O(1)$. (Not useful to have all items in one list.)
Example function.

\[ n = 4, \quad m = 16, \quad \text{hash table of size} \quad 4. \]

\[ h(x) = \frac{(ax + b) \mod 16 \div \lfloor \frac{(ax + b) \mod 16 + 1} {4} \rfloor} {4} \]

\[ a = 3, \quad b = 5 \]

\[ S = \{ 1, 2, 3, 12 \} \]

\[ h(S) = \{ \frac{8}{14}, \frac{11}{14}, \frac{10}{14}, \frac{9}{14} \} = \{ 2, 2, 2, 2 \} \]

All map to same location.

\[ a = 7, \quad b = 1 \]

\[ h(S) = \{ \frac{8}{14}, \frac{15}{14}, \frac{12}{14}, \frac{5}{14} \} \]

\[ = \{ 2, 3, 0, 1 \} \].
Let the array size be $n$.

Useful property: $\Pr \{ \exists x \mid h(x) = c \} \ll n^{-1}$

Of course, given a particular function $h$, we can always choose a set $S$ so that $h(x) = h(y)$ for all $x, y \in S$, where $|S| \ll n$.

So what we want is a chain of hash functions $h_1, h_2, \ldots, h_k$, s.t. a random $h_i$ is likely to distribute the elements of $S$ fairly evenly.

A sufficient condition for this (as we shall see) is that

$$
\Pr \{ h(x) = i \text{ and } h(y) = i \} \ll \frac{1}{n} \quad \forall x \neq y
$$

$$
\Pr \{ h(x) = h(y) \} \ll \frac{1}{n}
$$

For item $x$, how many items, on the average, are in the same bucket as $x$?

$E[\text{number of items in } h_i(x)]$

$$
\mathcal{E} = E\left[ \sum_{y \in S} 1_{h(y) = h(x)} \right]
= \sum_{y \neq x} E\left[ 1_{h(y) = h(x)} \right]
= \sum_{y \neq x} E\left[ 1_{h(y) = h(x)} \right]
= (|S| - 1) \tau \left( \frac{1}{|S|} \right) = \Theta \left( \frac{n - 1}{n} \right) = \Theta \left( \frac{n}{n} \right).
$$

If $\tau = \Theta(n)$ (for some $c > 1$), this is $\Theta \left( \frac{n}{c} \right)$ items.
Choosing \( c = 1 \) implies each operation takes \( O(1) \) time on average.

Some families of hash functions:

1. \( h(x) = (ax + b) \mod p \mod r \) where \( p, r \) is prime and \( 1 \leq a < p \)
   \( 0 \leq b < p \)

2. \( h(x) = \left\lfloor \frac{(ax + b) \mod 2^{i+j}}{2^j} \right\rfloor = \left\lfloor (ax + b) \mod 2^{i+j} \right\rfloor \div 2^j \)

where \( 2^i \geq m, 2^i \geq n \)
\( 1 \leq a < 2^{i+j} - 1 \)
\( 0 \leq b < 2^{i+j} - 1 \)

We analyze the second family.
Analyze

\[ \text{Prob} \left[ h(x) = c = h(y) \right] = \frac{1}{2^i} \quad \text{for } x \neq y \]

allowing \( a = 0 \) (only increases prob. of collision).

Want \( ax + b \mod 2^{ix_j} = c 2^j + r \) when \( 0 \leq r < 2^j \)

\( ay + b \mod 2^{iy_j} = c 2^j + s \) when \( 0 \leq s < 2^j \).

WLOG suppose that \( x \neq y \).

Then want \( a(x-y) \mod 2^{ix_j} = r-s \) \((*)\)

\[ \text{Case 1: } x-y \text{ is odd} \]

Then \( 0, (x-y), 1, (x-y), \ldots, (2^{iy_j}-1)(x-y) \mod 2^{ix_j} \) all distinct.

For \( j \) odd,

\[ a(x-y) = B(x-y) \mod 2^{ix_j} \]

i.e. \( (B-a)(x-y) = 1 \cdot 2^{ix_j} \) for some integer \( \lambda \)

\[ \Rightarrow 2^{ix_j} \uparrow B - a \Rightarrow B = a. \]

Thus there is exactly only value of \( a \) satisfying \((*)\).

\[ \therefore \# \text{ of choices of } (a, i, r, s) = 2^j. \]

In addition, need

\[ ax + b \mod 2^{ix_j} = c 2^j + r. \]

Just one choice of \( b \) given \( a, c, r. \)

Hence \( \text{Prob} \left[ h(x) = c = h(y) \right] = \frac{2^j}{2^{2(i-ix_j)}} = \frac{1}{2^i} \)

\( \therefore \text{chorus of } c : \)

Hence \( \text{Prob} \left[ h(x) = h(y) \right] = \frac{1}{2^i} \).
Case 2. \[ 2^h 1(x-y), \ 2^h \lambda_1(x-y), \ \alpha \leq h < j. \]

Then \[ 0(x-y), \ 1(x-y), \ldots, (2^h \cdot 2^{i-j-h}) (x-y) \] all distinct.

Thus if \[ 2^h + i < j \] there are 0 values satisfying (v).

If \[ 2^h | i < j \] there are \[ 2^h \] values satisfying (w) of the form:

\[ a, a + 2^h \cdot 2^{i-j-h}, a + 2^h \cdot 2^{i-j-h}, a + 2^h \cdot 2^{i-j-h}, \ldots, a + 2^h (\lambda_i) \cdot 2^{i-j-h}. \]

\[ \therefore \text{# of choices} \ (a, r, s) \ \text{is} \ 2^h \cdot 2^j \cdot 2^{i-j-h} = 2^{2j}. \]

Rest of argument as before.
Cloud Hashing

Store lists in the same table and avoid explicit pointers.

Instead, have

linear probing:

\[ h(x) \]

if full, search for first empty location after \( h(x) \) & store \( x \) there.

double hashing:

Use a second hash function \( d(x) \) to give

sequence of locations to try: \( h(x), h(x) + d(x), h(x) + 2d(x), \ldots \)

Harder to analyze.

Deleting are a nuisance:

\[ h(x) \]

\[ \Rightarrow \]

\[ h(b) \]

\[ \text{used} \]

\[ x \]

Need to match \( h(x) \) locations as used so knows to keep going if

- search \( (y) \) occurs.
Bloom Filters

A compact storage scheme, but now there is some probability of an 
incorrect answer.

Supports insertions & searches.

Initially all entries in table are \( \emptyset \).

When \( x \) is inserted set \( A[\text{hash}(x)] = 1 \).

Later, if searching, report \( x \) is in the table, set \( A[\text{hash}(x)] = 1 \).

So a "not in set" is always correct.

Chance of incorrect answer is \( 1 \leq 1 \),

is at most \( \frac{1}{e} \).

How can we improve the probability?

Use 2 tables \( 2 \) hash functions \( h_1, h_2 \).

Answer \( x \) is in set only if \( h_1(x) = 1 = h_2(x) \).

Prob of incorrect answer is at most \( \left( \frac{1}{e} \right)^2 \).

Note if before we store 64-bit items, now we using the same 
space we could have 2 tables of size 32n, for an error 
prob \( \frac{1}{1000} \), 4 tables of size 16 for an error prob 
\( \frac{1}{32,000} \).

There is a tradeoff in the number of hash function evaluations 
and the error probability.
Passwords

Same Idea

\[ p \Rightarrow h(p) \]

**pep:** compute & store \( h(p) \).

Additional requirement: given \( h(p) \) & \( h \), it should be hard to figure out \( p \).

Not true for the universal hash function, just shown.

---

We do not know how to prove such results at present.

It amounts to showing that inspired guessing is computationally hard.

In the end, this is what the question "\( P = NP? \)" amounts to.