Introduction to hash tables

Various data structures for mapping storage

A mapping relation is commonly used in dictionaries and databases, to store this relation, popular data structures include:

- Tree (Balanced trees, like AVL, Red Black, etc.)

Complexity	Average	Worst
Space	$O(n)$	$O(n)$
Search	$O(log\ n)$	$O(logn)$
Insert	$O(log\ n)$	$O(log\ n)$
Delete	$O(log\ n)$	$O(log\ n)$

- Skip-list

Complexity	Average	Worst
Space	$O(n)$	$O(n\ log\ n)$
Search	$O(log\ n)$	$O(n)$
Insert	$O(log\ n)$	$O(n)$
Delete	$O(log\ n)$	$O(n)$

- *Hash table*

Complexity	Average	Worst
Space	$O(n)$	$O(n)$
Search	$O(1)$	$O(n)$
Insert	$O(1)$	$O(n)$
Delete	$O(1)$	$O(n)$

Note: depending on the specific data we would like to hash, the time complexity of hashing itself alone will change, but typically we assume that it is constant time.

What is a hash table?

Hash tables have two core components:

1. A hash function $h : u \rightarrow \{0, \ldots, n-1\}$, which maps all elements in the value domain (universe) u to a *hash value* in image domain $\{0, \ldots, n-1\}$.
2. A set of n hash buckets, with unique indexes in image domain $\{0, \ldots, n-1\}$.

We may define the methods needed by a hash table to clarify its functions:

```
class HashTable:
    def __init__(self):
        # Pick a random hash function h
        pass

    def insert(self, key, value):
        id = h(key)
        # insert a tuple (key, value) in bucket #id
        pass

    def find(self, key):
        id = h(key)
        # find tuple with key=key in bucket #id
        return tuple

    def remove(self, key):
        id = h(key)
        # remove tuple (key, value) from bucket #id
        pass

    def update(self, key, value):
        self.remove(key)
        self.insert(key, value)
```

In this definition, we assume that every hash bucket can store multiple key/value entries, note that in a implementation using "probing", this assumption is not necessarily true.

Properties needed by a hash function

In order to have a hash table implementation with average insert / find / remove time complexity equal to $O(n)$, the chosen hash function should have some special properties and guarantees, otherwise it may put all elements in the same bucket, which reduces the hash table to a typical linked list.

For most cases, hash function h needs to have the following properties, we will see reasons behind these choices in sub-sections.

- Should be sufficiently random.
- Should be fast to compute.
- Should have concise description.

Formal definitions

Running time: Suppose the hash table contains keys $x_1, \ldots, x_m$, we insert a key y, running time is the relative run time against $Size(bucket_{h(y)})$.

Hash function: Hash function is a function which maps the key domain u to hashed image domain from 0 to $n-1$: $h : u \to \{0, \ldots, n-1\}$.

Random hash function: A random hash function h satisfies: $h \in_u \mathcal{H}$ (h is uniformly chosen from a family $\mathcal{H}$, probably with some random parameters).

Perfect hash function: A perfect hash function h maps every element x in the input set $s \subseteq u$ to hash domain with no collision, i.e. $\forall x_i, x_j \in s, h(x_i) \neq h(x_j)$.

Symbols: Let u denote the original domain, m denote the hash domain, $|u|$ and $|m|$ denote the size of these two domains, $n = |m|$ be the size of hash domain and number of hash buckets.

Deterministic

If h is deterministic can it be a perfect hash function?

1. If $|u| > n$
   If we hash all elements in u, and put them to buckets, then one bucket will contain $\lceil \frac{|u|}{n} \rceil$ elements, then an adversary can pick all elements belonging to this bucket, then for a set s made up of these elements, h is no longer perfect.
2. if $|u| \leq n$
   Then it is possible, because all elements in the universe can be mapped to a unique bucket. (Possible use case: counting sort / Storing json, html files used in browsers with almost fixed static keys, sub-graph storage in graph traversing is also a possible use case.)

A perfect hash function satisfying $|s| = n$ is called a minimal perfect hash function.

And typically perfect hash functions are deterministic, although there exists dynamic perfect hashing methods, they are typically very complex, a usual substitute for these dynamic perfect hashing functions is "cuckoo hashing"

Random oracle model

A ROM hash function satisfies:

1. $h_R(x)$ is a random hash function.
2. $h_R(x)$ returns a random number for each $x \in u$.
3. All hashed values $h(x_0)$ and $h(x_1)$ are mutually **independent**, i.e. subject to $i.i.d.$.

Explanations:

- *Why randomly choose the function?*

  Because we need to evaluate the performance of family $\mathcal{H}$ by evaluating their $\mathbb{E}(run\ time)$.

  And for some security reasons, because we do not want adversaries be able to infer the original value from the hashed value.

- Randomness of $h \nRightarrow i.i.d.$

  The randomly chosen h (through parameter R) does not guarantee $i.i.d.$.

  Suppose the mapped domain is {0, 1}, and there is $\forall R\ \forall x \in u, h_R(x) = 0$, then:

  $$P\{h(x_0) = 0\} \neq P\{h(x_0) = 0 \mid h(x_1) = 1\} * P\{h(x_1) = 1\}$$

  Because the right side (RHS) of the equation is always 0 but left is always 1.

Universal hash functions

A universal hash function, usually prefixed by k, reading "k-universal hash functions", satisfies:

1. $h_R(x)$ is a random hash function.
2.

$$\forall \{x_1, \dots, x_k\} \subseteq u, (y_1, \dots, y_k) \in m \qquad P_{h_R \in \mathcal{H}}\{h(x_1) = y_1 \wedge\ \dots\ \wedge h(x_k) = y_k\} \leq |m|^{-k}$$

Note that all x are distinct and y are not necessarily distinct.

For 2-universal hash functions, we usually use an even weaker requirement: $P_{h_R \in \mathcal{H}}\{h(x_1) = h(x_2)\} \le \frac{1}{|m|}$ for the sake of simplicity, this should be trivial to prove using the above definition.

Relation between above assumptions

So there are 3 levels of assumptions in random hash functions:

1. h is a perfect hash function (every element in its own bucket), $|u| \ll n$.
2. h is a ROM.
3. h is a 2-universal hash function.

The strongness relation is: 1 > 2 > 3.

$1 \Rightarrow 2$ is trivially understandable, so we will skip it now.
$2 \Rightarrow 3$: suppose h is a ROM, we can prove assumption 3 by:

$$
\begin{aligned}
&P\{h(x) = h(y)\} \\
&= \sum_{k=0}^{n-1} P\{h(x) = k | h(y) = k\} \\
&= \sum_{k=0}^{n-1} P\{h(x) = k\} * P\{h(y) = k\} \\
&= 1/n
\end{aligned}
$$

2-universal hash function performance

Running time for a hash table with 2-universal hash function is constant.

Proof
Suppose hash table contains m elements: $x_1, \dots x_m \in u$, and $m \le n$, we search for element $y \in u$:

$$
\begin{aligned}
&\mathbb{E}\{runtime\} \\
&\text{Note: c is a constant like O}(1)\text{, used to represent other costs like hashing} \\
&\le c * \mathbb{E}(\sum_i^m \mathbb{I}\{h(x_i) = h(y)\}) \\
&\text{Note: } \mathbb{E}(\mathbb{I}\{h(x_i) = h(y)\}) = P\{h(x_i) = h(y)\} \\
&= c * \sum_i^m \mathbb{E}(\mathbb{I}\{h(x_i) = h(y)\}) \\
&\text{Note: } P\{h(x_i) = h(y)\} \le \frac{1}{n} \text{ if } x_i \ne y \\
&\le c * (\frac{m-1}{n} + 1) \\
&\text{Note: if } m \le n \\
&\le 2 * c
\end{aligned}
$$

If $m > n$, rehashing happens, i.e. grow bucket size n, update hash function to accommodate new bucket size, then compute a new hash value for each element in the hash table, and finally delete the old table. Typically we will grow n exponentially, e.g.: to $2n$.

Rehashing cost is $O(n)$ in total, and average runtime cost for each element is still $O(1)$ after dividing the total cost by element number, this can be proved by:

> Suppose grow hash table from 1 to M:
>
> - Insertion cost: From element 1 to M, constant for each so $O(M)$ in total.
> - Re-hashing cost: From 2, 4, 8, ..., to M: $2+4+8+...M \leq 2M$, so total cost is $O(M)$.
> - Deletion cost: $O(M)$

After rehashing, $m \leq n$ is satisfied again, so running time will still be $O(1)$ after that.