Hash table

A hash table is a data structure that maps keys to values. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. The total worst-case time complexity of performing any operation is \(O(n)\), where \(n\) is the number of keys stored in the table.

Cuckoo hashing

In Cuckoo hashing, each key is associated with two possible positions. On average, this results in \(O(1)\) worst-case time per operation. However, it does mean that the amortised expected worst-case time complexity of an operation is \(O(1)\).

Dynamic perfect hashing

A dynamic dictionary stores (key, value) pairs and supports:

- add(key, value), lookup(key) (which returns value) and delete(key)

In the Cuckoo hashing scheme:

- Every lookup and every delete takes \(O(1)\) worst-case time.
- The space is \(O(n)\) where \(n\) is the number of keys stored.
- An insert takes amortised expected \(O(1)\) time.

What does amortised expected \(O(1)\) time mean?

let's build it up...

\(O(1)\) expected time per operation means every operation takes constant time in expectation.

The total expected time complexity of performing any \(n\) operations is \(O(n)\)*

This does not imply that every operation takes constant time in expectation.

However, it does mean that the amortised expected time complexity of an operation is \(O(1)\).

In Cuckoo hashing there is a single hash table but two hash functions: \(h_1\) and \(h_2\).

Each key in the table is either stored at position \(h_1(x)\) or \(h_2(x)\).

Therefore, as claimed, lookup takes \(O(1)\) time... but how do we do inserts?

**THEOREM**

Cuckoo hashing is a scheme:

- supports:
  - \(O(1)\) worst-case lookup time,
  - \(O(1)\) amortised expected worst-case time per operation.

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.

If for any \(n\) arbitrary operations arrive online, one at a time:

- The space is \(O(n)\) where \(n\) is the number of keys stored.
- An insert takes amortised expected \(O(1)\) time.

Using weakly universal hashing:

For any \(n\) operations, the expected run-time is \(O(1)\) per operation.

In fact this result can be generalised...
We make the following assumptions:

- \( h_1 \) and \( h_2 \) are independent. i.e. \( h_1(x) \) says nothing about \( h_2(x) \), and vice versa.
- \( h_1 \) and \( h_2 \) are truly random. i.e. each key is independently mapped to a particular position in the hash table with probability \( \frac{1}{m} \).

Computing the value of \( h_1(x) \) and \( h_2(x) \) takes \( O(1) \) worst-case time.

There are at most \( n \) keys in the hash table at any time.

---

**Rehashing**

If we fail to insert a new key \( x \), (i.e. we still have an "evicted" key after moving around keys \( n \) times) then we declare the table "rubbish" and rehash.

What does rehashing involve?

Suppose that the table contains the \( k \) keys \( x_1, \ldots, x_k \) at the time of we fail to insert key \( x \).

To rehash:

- Randomly pick two new hash functions \( h_1 \) and \( h_2 \). (More about this in a minute.)
- Build a new, empty, hash table of the same size.
- Reinsert the keys \( x_1, \ldots, x_k \) and then \( x \), one by one, using the normal add operation.

If we fail while rehashing...we start from the beginning again.

This is rather slow...but we will prove that it happens rarely.

---

**Assumptions**

We will follow the analysis in the paper Cuckoo hashing for undergraduates, 2006, by Rasmus Pagh (see the link on unit web page).

We make the following assumptions:

- \( h_1 \) and \( h_2 \) are independent. i.e. \( h_1(x) \) says nothing about \( h_2(x) \), and vice versa.
- \( h_1 \) and \( h_2 \) are truly random. i.e. each key is independently mapped to a particular position in the hash table with probability \( \frac{1}{m} \).

Computing the value of \( h_1(x) \) and \( h_2(x) \) takes \( O(1) \) worst-case time.

There are at most \( n \) keys in the hash table at any time.
For any positions \( i \) and \( j \), and any constant \( c > 1 \), if \( m \geq 2cn \) then the probability that there exists a shortest path in the cuckoo graph from \( i \) to \( j \) with length \( \ell \geq 1 \), is at most \( \frac{1}{c^\ell m} \).

**Lemma**

**Proof by induction.**

**Base case:** \( \ell = 1 \).

Let \( K \) be the set of keys in the hash table. \( |K| \leq n \).

The probability that a key \( x \) is mapped to positions \( i \) and \( j \), i.e., either \( h_1(x) = i \), \( h_2(x) = j \) or \( h_1(x) = j \), \( h_2(x) = i \), is at most \( \frac{2}{m^2} \) (recall we have assumed independence between \( h_1 \) and \( h_2 \)).

Therefore (using the union bound) the probability that there is an edge between \( i \) and \( j \) is at most \( \frac{2n}{m^2} \leq \frac{1}{c m} \cdot \frac{1}{cn} \) since \( m \geq 2cn \).

**LEMMA**

**Proof continued...**

**Inductive step:** assume lemma is true for lengths \( 1, 2, \ldots, \ell - 1 \).

If there is a path between \( i \) and \( j \) of length \( \ell \) but no path shorter than \( \ell \), then there must be a position \( k \) such that:

- **A** there is a shortest path of length \( \ell - 1 \) from \( i \) to \( k \) that does not go through \( j \), and
- **B** there is an edge from \( k \) to \( j \).

By the inductive hypothesis, \( \Pr(A) \leq \frac{1}{c^\ell - 1} \).

Observe: The "not go through \( j \)" can only make the probability smaller.

Given that \( A \) is true, the probability that \( B \) holds is as follows:

\[
\Pr(A \land B) = \Pr(A) \cdot \Pr(B \mid A) \leq \frac{1}{c^\ell - 1} \cdot \frac{1}{m^2} = \frac{1}{m^2 c^\ell - 1}.
\]

The union bound over all "midpoints" \( k \) gives an upper bound on the probability of a shortest path between \( i \) and \( j \) of length \( \ell \):

\[
\sum_{k \in K} \Pr(A \land B) \leq \frac{1}{m c^\ell - 1} = \frac{1}{m^{1/c^\ell - 1}}.
\]

**Back to buckets**

We say that two keys \( x, y \) are in the same bucket (conceptually) if there is a path between \( h_1(x) \) and \( h_2(y) \) in the cuckoo graph.

For two distinct keys \( x, y \), the probability that they are in the same bucket is at most

\[
4 \sum_{\ell=1}^{\infty} \frac{1}{c^\ell - m} = \frac{4}{m(c-1)} = O\left(\frac{1}{m}\right)
\]

where \( c > 1 \) is a constant.

(another union bound over all possible path lengths.)

The time for an operation on \( x \) is bounded by the number of items in the bucket. (Assuming there are no cycles.)

So we have that the expected time per operation is \( O(1) \) (assuming that \( m \geq 2cn \)).

Further, lookups take \( O(1) \) time in the worst case.

**Hash table**

For any positions \( i \) and \( j \), and any constant \( c > 1 \), if \( m \geq 2cn \) then the probability that there exists a shortest path in the cuckoo graph from \( i \) to \( j \) with length \( \ell \geq 1 \), is at most \( \frac{1}{c^\ell m} \).

The probability that there is at least one cycle is at most \( \frac{m}{m(c-1)} = \frac{1}{c-1} \) (union bound over all \( m \) positions in the table.)

If we set \( c = 3 \), the probability is at most \( \frac{1}{4} \) that a cycle occurs (that there is a rehash) during the \( n \) insertions.

The probability that there are two rehashes is therefore \( \frac{1}{4} \), and so on.

So the expected number of rehashes during \( n \) insertions is at most \( \sum_{i=1}^{\infty} \left(\frac{1}{4}\right)^i = 1 \).

**Rehashing**

The previous analysis on the expected running time holds when there are no cycles.

However, we would expect there to be cycles every now and then, causing a rehash.

**How often does this happen?**

For simplicity, let us assume that there are already \( n \) keys in the table and we want to insert another \( n \) keys.

We assume now that the table size \( m \geq 4cn \), where \( c > 1 \) is the constant from the previous slides.

A cycle is a path from a vertex \( i \) back to itself, so use previous result with \( i = j \)...

**Lemma**

For any positions \( i \) and \( j \), and any constant \( c > 1 \), if \( m \geq 2cn \) then the probability that there exists a shortest path in the cuckoo graph from \( i \) to \( j \) with length \( \ell \geq 1 \), is at most \( \frac{1}{c^\ell m} \).

**Rehashing**

If the expected time for one rehash is \( O(n) \) then the expected time for all rehashes is also \( O(n) \) (this is because we only expect there to be one rehash).

Therefore the amortised expected time for the rehashes over the \( n \) insertions is \( O(1) \) per insertion (i.e. divide the total cost with \( n \)).

**Why is the expected time per rehash \( O(n) \)?**

First pick a new random \( h_1 \) and \( h_2 \) and construct the cuckoo graph using the at most \( n \) keys.

Check for a cycle in the graph in \( O(n) \) time (and start again if you find one) (you can do this using breadth-first search)

If there is no cycle, insert all the elements, this takes \( O(n) \) time in expectation (as we have seen).
A word about the assumptions

We have assumed true randomness. As we have discussed, this is not realistic.

We have seen that weakly universal hash families are realistic.

We can define stronger hash families with $k$-independence.

By changing the cuckoo hashing algorithm to perform a rehash after $k = \log n$ moves,

it can be shown (via a similar but harder proof) that the results still hold.

**Theorem**

In the Cuckoo hashing scheme:

- Every lookup and every delete takes $O(1)$ worst-case time,
- The space is $O(n)$ where $n$ is the number of keys stored,
- An insert takes amortised expected $O(1)$ time.