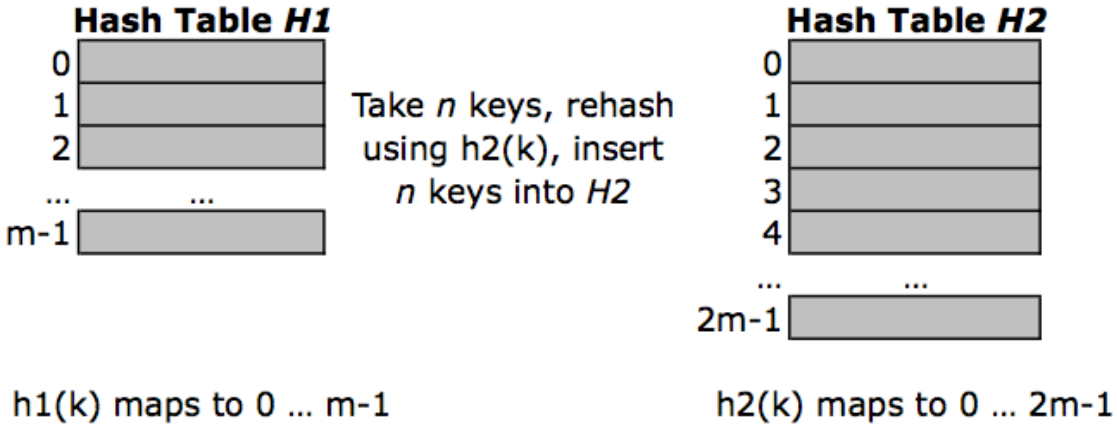
Hashing Part 2!

Resizing Hash Tables

Hash tables perform well if the number of elements in the table remain proportional to the size of the table. If we know exactly how many inserts/deletes are going to be performed on a table, we would be able to set the table size appropriately at initialization. However, it is often the case that we won't know what series of operations will be performed on a table. We must have a strategy to deal a various number of elements in the hash table while preserving an average $O(1)$ access, insertion, and removal operations.

To restrict the load balance so that it does not get too large (slow search, insert, delete) or too small (waste of memory), we will increase the size of the hash table if it gets too full and decrease the size of the hash table if it gets too empty.

Resizing a hash table consists of choosing a new hash function to map to the new size, creating a hash table of the new size, iterating through the elements of the old table, and inserting them into the new table.



Consider a hash table that resolves collisions using the chaining method. We will double the size of the hash table whenever we make an insert operation that results in the load balance exceeding 1, i.e. $n > m$. We will halve the size of the hash table whenever we make a delete operation that results in the load balance falling beneath $\frac{1}{4}$, i.e. $n < \frac{m}{4}$. In the next sections, we will analyze this approach and show that the average runtime of each insertion and deletion is still $O(1)$, even factoring in the time it takes to resize the table.

Increasing Table Size

After doubling the table size due to an insert, $n = \frac{m}{2}$ and the load balance is $\frac{1}{2}$.We will need at least $\frac{m}{2}$ insert operations before the next time we double the size of the hash table. The next resizing will take $O(2m)$ time, as that's how long it takes to create a table of size $2m$.

	0.5m insertions before resize				resize
Operation	insert	insert	...	insert	insert + resize
Runtime	*O(1)*	*O(1)*	...	*O(1)*	*O(2m)*

Redistribute *O(2m)* resize cost over 0.5m insertions

	0.5m insertions before resize				resize
Operation	insert	insert	...	insert	insert + resize
Runtime	*O(1)*	*O(1)*	...	*O(1)*	*O(1)*
Amortized Cost	*O(2m/0.5m)*	*O(2m/0.5m)*	...	*O(2m/0.5m)*	-

On average, since the number of elements is proportional to the size of the table at all times, each of the $\frac{m}{2}$ inserts before resizing will still take $O(1)$ time. The last insert will take $O(2m)$ time as we need to factor in the time it takes to resize the table. We can use amortized analysis to argue that the average runtime of all the insertions is $O(1)$. The last insert before resizing costs $O(2m)$ time, but we needed $\frac{m}{2}$ inserts before actually paying that cost. We can imagine spreading the $O(2m)$ cost across the $\frac{m}{2}$ inserts evenly, which adds an additional average amortized cost of $O(\frac{2m}{0.5m})$ per insert, or $O(1)$ per insert. Since the cost of insertion before was $O(1)$, adding an additional $O(1)$ amortized cost to each insert doesn't affect the asymptotic runtime and insertions on average take $O(1)$ time still.

Decreasing Table Size

Similarly, after halving the table size due to an deletion, $n = \frac{m}{2}$. We will need at least $\frac{m}{4}$ delete operations before the next time we halve the size of the hash table. The cost of the next halving is $O(\frac{m}{2})$ to make a size $\frac{m}{2}$ table.

The $\frac{m}{4}$ deletes take $O(1)$ time and the resizing cost of $O(\frac{m}{2})$ can be split evenly across those $\frac{m}{4}$ deletes. Each deletion has an additional average amortized cost of $O(\frac{0.5m}{0.25m})$ or $O(1)$. This results in maintaining the $O(1)$ average cost per deletion.

Rolling Hash

Rolling hash is an abstract data type that maintains a list and supports the following operations:

- `hash()`: computes a hash of the list.
- `append(val)`: adds `val` to the end of the list.
- `skip(val)`: removes the front element from the list, assuming it is `val`.

In the case of strings, the list is a list of characters.

Data Structure - Algorithms for Each Operations

Characters can be interpreted as integers, with their exact values depending on what type of encoding is being used (e.g. ASCII, Unicode). For example the ASCII code for 'A' is 65 and 'B' is 66. This means we can treat strings as lists of integers.

Key idea: treat a list of items as a multidigit number u in base a ('concatenate' list items into a big number). For example, we can choose $a = 256$, the alphabet size for ASCII code.

- `hash()`: u mod p for prime p (division method).
- `append(val)`: $((u \cdot a) + val)$ mod p = $[(u \bmod p) \cdot a + val] \bmod p$
- `skip(val)`: $[u - val \cdot (a^{|u|-1} \bmod p)] \bmod p = [(u \bmod p) - val \cdot (a^{|u|-1} \bmod p)] \bmod p$

Hashing intuition: choose $a = 100$ for easy illustration purpose, and $p = 23$. hash([61, 8, 19, 91, 37]) = (6108199137 mod 23) = 12. In general, hash($[d_3, d_2, d_1, d_0]$) $= d_3 \cdot a^3 + d_2 \cdot a^2 + d_1 \cdot a^1 + d_0 \cdot a^0) \bmod p$.

Sliding intuition:

- list: [3, 14, 15, 92, 65, 35, 89, 79, 31]
- from [3, 14, 15, 92, 65] to [14, 15, 92, 65, 35], we get hash values from 11 to 6.
- from [14, 15, 92, 65, 35] to [15, 92, 65, 35, 89], we get hash values from 6 to 5.

Fast rolling hash:

- Cache the result u mod p.
- Need to avoid exponentiation in skip: cache $a^{|u|-1}$ mod p (skip multiplier).
  - append: multiply it by base
  - skip: divide it by base (division is expensive, can use multicative inverse).

Good Functions

There are four main characteristics of a good hash function:

1. The hash value is fully determined by the data being hashed.
2. The hash function uses all the input data.
3. The hash function "uniformly" distributes the data across the entire set of possible hash values.
4. The hash function generates very different hash values for similar strings.

Let's examine why each of these is important.

- Rule 1: If something else besides the input data is used to determine the hash, then the hash value is not as dependent upon the input data, thus allowing for a worse distribution of the hash values.
- Rule 2: If the hash function doesn't use all the input data, then slight variations to the input data would cause an inappropriate number of similar hash values resulting in too many collisions.
- Rule 3: If the hash function does not uniformly distribute the data across the entire set of possible hash values, a large number of collisions will result, cutting down on the efficiency of the hash table.
- Rule 4: In real world applications, many data sets contain very similar data elements. We would like these data elements to still be distributable over a hash table.

Bad Functions

Credit card numbers

Credit card numbers have structure. The first 4 digits in the 16-digit number are the bank number, so using them in a hash function can yield collisions – for example, MIT students most likely have cards issued by the few banks in the area (MITFCU, Bank of America MA, etc.). Also, the last digit is a checksum digit, designed so that most typos can be corrected offline, without querying a bank server, so a hash function that uses the last digit wastes time without gaining any additional randomness.

Notes on Implenting Hash Function in Python

Pre-hashing is done by calling the `__hash__` instance method on an object. When overriding `__eq__` you should also override `__hash__`, so that objects can be used as keys in a dictionary.