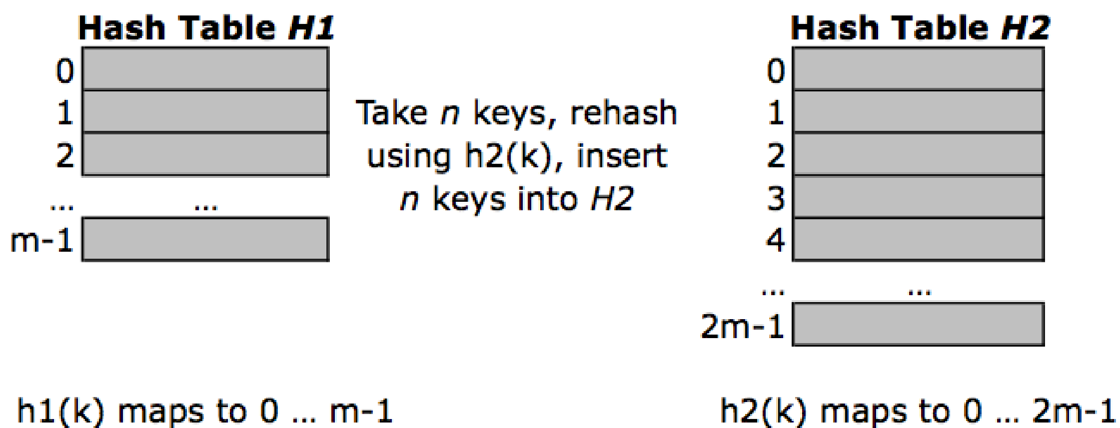
Resizing Hash Tables

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.

Performance of Open Addressing

Recall that searching, inserting, and deleting an element using open addressing required a probe sequence (e.g. linear probing, quadratic probing, double hashing). To analyze the performance of operations in open addressing, we must determine on average how many probes does it take before we execute the operation. Before, we made the **simple uniform hashing assumption (SUHA)**, which meant a hash function mapped to any slot from 0 to $m - 1$ with equal probability. Now, we make the **uniform hashing assumption (UHA)**, which is a slight extension from SUHA. UHA assumes that the probe sequence is a random permutation of the slots 0 to $m - 1$. In other words, each probe looks likes we're examining a random slot that we havent examined before.

If the table has load balance α, that means there is a $p = 1 - \alpha$ probability that the first probe will find an empty slot under UHA. If the first probe is a collision, note that the probability that the

second probe will find an empty slot is greater than p, since there are an equal number of empty slots that we could insert in, but were choosing randomly from a pool of fewer slots. In general, after each collision, there is a probability of at least p that we will probe into an empty slot.

Using principles of probability, if there is exactly a probability of p that we will find an empty slot at each probe, then we expect to probe $\frac{1}{p}$ times before we succeed. For example, if $p = \frac{1}{4}$, we expect to probe 4 times before we find an empty slot. Since in our case, our probability of success is actually increasing after each probe, $\frac{1}{p}$ is a high estimate on how many times we probe before we succeed. Since $p = 1 - \alpha$, we expect to probe at most $\frac{1}{p}$ times. Looking at the behavior of the $\frac{1}{1-\alpha}$ graph, it is clear that with open addressing, performance is fairly good until α approaches too close to 1.

Universal Hashing

With a fixed hashing function, an adversary could select a series of keys to insert into the hash table that all collide, giving the hash table worst case performance. Universal hashing is the idea that we select the hash function randomly from a group of hash functions. This means an adversary cannot choose keys that he knows will give worst case performance anymore, since the adversary doesn't even know what hash function will be chosen for the table. If we form the group of hash functions carefully, we can assure that the expected time for each operations is $O(1)$, even if there is an adversary who is trying to achieve worst case performance.

For universal hashing to work, the group of hash functions H must be **universal**. This means that for each pair of distinct keys k, l, in the universe of keys, the number of hash functions in the group for which $h(k) = h(l)$ is at most $\frac{|H|}{m}$. This means, for each pair of distinct keys, the chances of picking a hash function in which they collide is at most $\frac{1}{m}$, which is the same probability given by the simple uniform hashing assumption.