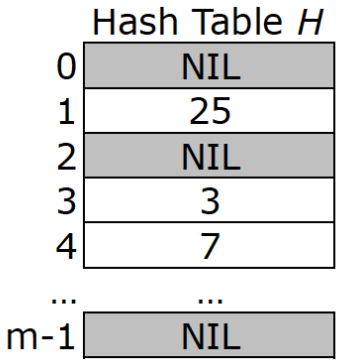
Overview of Hash Tables

A hash table is a data structure that supports the following operations:

- `insert(k)` - puts key k into the hash table
- `search(k)` - searches for key k in the hash table
- `remove(k)` - removes key k from the hash table

In a well formed hash table, each of these operations take on average $O(1)$ time, making hash tables a very useful data structure.

You can think of a hash table as a list of m slots. Inserting a key puts it in one of the slots in the hash table, deleting a key removes it from the slot it was inserted in, and searching a key looks in the slot the key would have been inserted into to see if it is indeed there. Empty slots are designated with a NIL value. The big question is figuring out which slot should a key k be inserted into in order to maintain the $O(1)$ runtime of these operations.



Hash Functions

Consider a function $h(k)$ that maps the universe U of keys (specific to the hash table, keys could be integers, strings, etc. depending on the hash table) to some index 0 to m. We call this function a **hash function**. When inserting, searching, or deleting a key k, the hash table hashes k and looks at the $h(k)$th slot to add, look for, or remove the key.

A good hash function

- satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots. The hash function shouldn't bias towards particular slots
- does not hash similar keys to the same slot (e.g. compiler's symbol table shouldn't hash variables `i` and `j` to the same slot since they are used in conjunction a lot)
- is quick to calculate, should have $O(1)$ runtime
- is deterministic. $h(k)$ should always return the same value for a given k

Example 1: Division method

The division method is one way to create hash functions. The functions take the form

$$h(k) = k \bmod m \tag{1}$$

Since we're taking a value mod m, $h(k)$ does indeed map the universe of keys to a slot in the hash table. It's important to note that if we're using this method to create hash functions, m should not be a power of 2. If $m = 2^p$, then the $h(k)$ only looks at the p lower bits of k, completely ignoring the rest of the bits in k. A good choice for m with the division method is a prime number (why are composite numbers bad?).

Example 2: Multiplication method

The multiplication method is another way to create hash functions. The functions take the form

$$h(k) = \lfloor m(kA \bmod 1) \rfloor \tag{2}$$

where $0 < A < 1$ and $(kA \bmod 1)$ refers to the fractional part of kA. Since $0 < (kA \bmod 1) < 1$, the range of $h(k)$ is from 0 to m. The advantage of the multiplication method is it works equally well with any size m. A should be chosen carefully. Rational numbers should not be chosen for A (why?). An example of a good choice for A is $\frac{\sqrt{5}-1}{2}$.

Collisions

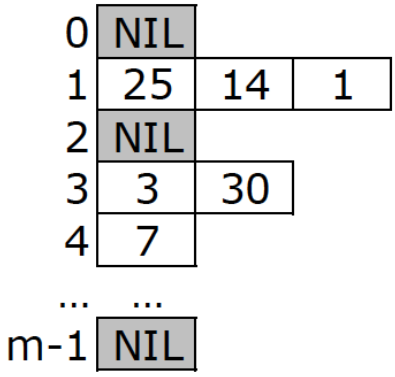
If all keys hash to different slots, then the hash table operations are as fast as computing the hash function and changing or inspecting the value of an array element, which is $O(1)$ runtime. However, this is not always possible. If the number of possible keys is greater than the number of slots in the hash table, then there must be some keys that hash into the same slot, in other words a **collision**. There are several ways to resolve a collision.

Chaining

In the chaining method of resolution, hash table slot j contains a linked list of every key whose hash value is j. The hash table operations now look like

- `insert(k)` - insert k into the linked list at slot $h(k)$
- `search(k)` - search for k in the linked list at slot $h(k)$ by iterating through the list
- `remove(k)` - search for k in the linked list at slot $h(k)$ and then remove it from the list

With chaining, if a key collides with another key, it gets inserted into the same linked list in the slot they hash into.



In the ideal case, all keys hash to different slots and every linked list has at most 1 element, keeping the runtimes of the operations at $O(1)$. In the worst case, all n keys inserted into the hash table hashes to the same slot. We then get a n size linked list which takes $O(n)$ to search through, resulting in $O(n)$ search and remove. This is why choosing a hash function that equally distributes keys to all slots is important.

If there are n keys in a hash table with m slots, we call the **load factor** α for the hash table to be $\frac{n}{m}$. Under the assumption of simple uniform hashing, the length of each linked list in the hash table is α. As long as the number of keys inserted is proportional to the size of the hash table, $\alpha = O(1)$, thus the operations on average are $O(1)$ as well.

Open Addressing Collisions

A hash table may use **open addressing**, which means that each slot in the hash table contains either a single key or NIL to indicate that no key has been hashed in that slot. Unlike chaining, we cannot fit more than one key in a single slot, so we must resolve collisions in a different way. We must have a method to determine which slot to try next in the case of a collision. We still try to put a key k into slot $h(k)$ first, but if that slot is occupied, we keep trying new slots until we find an empty one to put the key into.

Linear probing resolves collisions by simply checking the next slot, i.e. if a collision occurred in slot j, the next slot to check would be slot $j + 1$. More formally, linear probing uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m \tag{3}$$

Where $h'(k)$ is the hash function we try first. If $h(k, 0)$ results in a collision, we increment i until we find an empty slot. One drawback to linear probing is if keys hash to slots close to each other, a cluster of adjacent slots get filled up. When trying to insert future keys into this cluster, we

must then traverse through the entire cluster in order to find an empty slot to insert into, which can slow down our hash table operations.

Quadratic probing resolves collisions in a similar fashion:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \tag{4}$$

for some constants c_1, c_2. Instead of linearly traversing through the hash table slots in the case of collisions, quadratic probing introduces more spacing between the slots we try in case of a collision, which reduces the clustering effect seen in linear probing. However, a milder form of clustering can still occur, since keys that hash to the same initial value will probe the exact same sequence of slots to find an empty slot.

Double hashing resolves collisions by using another hash function to determine which slot to try next:

$$h(k,i) = (h_1(k) + i h_2(k)) \bmod m \tag{5}$$

With double hashing, both the initial probe slot and the method to try other slots depend on the key k, which further reduces the clustering effect seen in linear and quadratic probing.

Searching for a key in a hash table using open addressing involves probing through slots until we find the key we want to find or NIL. If we encounter a slot with a NIL value before finding the key itself, that means that the key in question is not in the hash table.

Deleting for a key involves searching for the key first. Once the key to be deleted is found, we remove it by replacing the key in that slot with a dummy DELETED value. Note that we cannot replace the key with a NIL value, or else searching for keys further down in the probe sequence will falsely return NIL. We must replace it with a dummy value indicating that a key was once present in this slot, but not anymore.