The half ass introduction to Hash Tables

Synopsis

Most programming languages provide the Map and Hash Table data structures in their standard API. We are going to study the principles behind the latter from the perspective of a programmer about to use on in their own projects. We will then take a look at how a simplified version of such a data structure might be implemented from scratch.

Disclaimer

This document provides elementary information about the concept of Hash Table. It is suitable as an introduction for those who were never exposed to it but falls short of providing a complete analysis of algorithmic performance and a treatment of the topic of Collisions.

No actual donkeys were armed in the process of writing this tutorial.

The idea of “Dictionary”

One of the easiest ways to illustrate what a hash table does is to start with the idea of Associative Maps and their archetypal application to implementing dictionary-like data structures.

The Hash Table API

Simply put, we want to be able to store in a data structure pairs of information { k, v } where k represents a key used to access an associated value v. To each unique key K, there is only one value associated in the hash table. The order in which these pairs are stored is irrelevant since we will access values based on their keys.

This differentiates hash tables from the data structures we previously discussed. Instead of retrieving a given value by specifying its index in an array or its position in a linked list, we provide a key. This key might be of any arbitrary data type; e.g. integer value, floating point value, string, structure…

To give you a pragmatic overview of how this data structure might be used, let’s establish its API.

Operation	Parameters		Returns	Role
PUT	H K V	hastable key value	1 – success 0 – key k was already in H	Adds { K,V} to the hash table H. Do nothing if there is already any element { K , ? } in H
REMOVE	H K	hash table key	1 – success 0 – key k wasn’t found in H	Remove { K , V } from H if found
GET	H K	hash table key	V if { K, V} in H, NULL otherwise	Returns the value associated with key K in hash table H

You will note that we’re not making any assumptions on what data types are used to implement the keys and values. We’ve already seen that a key might be of any data type. The same applies for the values.

We might end up with a hash table associating words in a given language to their translation just as we might end up with one associating social security numbers with employee records data structures.

Using a Hash Table

Let’s illustrate the usage of a Hash Table by looking at how we would populate one with information, step by step. For our example, we will assume that we want to store translations of words in such a data structure. Our keys will therefore be strings and so will our values. The keys will be used as the words to be translated and will allow us to retrieve their translation from the hash table by a GET operation resulting in the associated value. Notice that the hash table doesn’t offer a symmetrical functionality; while we might obtain the unique value associated to any given key; we are not able to obtain a key given a specific value. The data structure doesn't even guarantee that there are not multiple different keys associated to the same value. That's not how a hash table works.

So, let's assume that we already created and initialized an empty hash table H. We will represent the contents of H as an unordered list of { K, V} pairs. This notation will allow us to understand how the API is used without looking yet into implementation details.

Operation	Parameters	Returns	Inside of H
			H = { }

PUT	K = "wolf"
	V = "loup"
		1	H = { {"wolf" , "loup"} }

PUT	K = "dog"
	V = "chien"
		1	H = { {"wolf" , "loup"} , { "dog" , "chien" } }

PUT	K = "woof"
	V = "woof"
		1	H = { {"wolf" , "loup"} , { "dog" , "chien" } {"woof" , "woof"} }

PUT	K = "woof"
	V = "wouf"
		0	Same

REMOVE	K ="wouf"	0	Same
REMOVE	K = "woof"	1	H = { {"hello" , "salut"} , { "dog" , "chien" } }

PUT	K = "woof"
	V = "wouf"
		0	H = { {"hello" , "salut"} , { "dog" , "chien" } {"woof" , "wouf"} }

GET	K = "wolf"	"loup"	Same
GET	K = "loup"	NULL	Same

This series of operations on H illustrates the entire API and the situations where its functions operate successfully or not. Now that we have a better grasp on what the data structure might be able to do for us and how we would use it, let's talk about its implementations.

Implementing Hash Tables

Implementing a hash table as a linked list?

When defined like so, this data structure seems trivial; let's simply store the pairs { K, V } as a linked list or in a dynamically resizable array. That would work just fine. However, there is one more feature of a hash table that we didn't mention so far; Performance.

In the above example, if H is represented as a linked list, the PUT operation is trivial. Simply add the new {K, V} pair at the end of the linked list. Such an insertion takes an amount of operations independent of the number of elements in the list if you think about keeping a pointer to its last element in addition to a pointer to its first element. The GET and REMOVE operations, on the other hand, will not be quite as fast… Since a hash table is not ordered, we were able to insert very efficiently new pairs. Retrieving them will be much more involved requiring us to iterate over all the elements until we find a pair with the key we are looking for. At best, we find it when looking at the first element. At worst, we'll have to look at all elements before to realize it's not there.

That is not what is expected of a hash table. The GET operation has to be optimized so that, ideally, we are able to retrieve a value from its key in an amount of time which remains constant regardless of the number of elements already stored in the hash table.

A first alternative would be to maintain the hash table as a *sorted* linked list. Retrieval time, while better, will not yet reach the results we would be able to obtain with a sorted array; i.e. logarithmic time with a binary search. The key is *direct* access. While we need to iterate over all of the first n-1 elements of a linked list in order to access element # n, an array allows us to access directly that element using n as an index. There you go, we need to implement hash tables as some sort of array in order to get closer to the ideal direct access to each element through the key.

Implementing a hash table as an array?

So, our hash table will store the { K , V } pairs in an array. No problems. However, we still have to iterate over this entire unordered array every time we are looking for a value V based on its key K. Sure, we might order the elements of the array based on their value for K… Wouldn't it be just better if we could somehow use K itself as an index in the array to access directly the { K, V } pairs?

This is exactly what hash tables are about. Let's define our hash table as an array of references to { K , V } pairs or arbitrary size **N**. The indexes of this array range in [0..N-1]. We need to define a function which will convert any key K into a value H(K) within that range so that we can then use H(K) as an index in the hash table array. Such a function is referred to as a *hash function* or *hashing function*. It is, along with an array of pairs, referred to as the *buckets*, one of the two main pieces of a hash table's implementation. Next section will provide you with a primer on hash functions, knowing that there is much more there that has been studied to fit in this guide.

Hash Functions

What do they look like?

We've already mentioned several times that a key might be of any data type. The hash function is designed to convert a specific range of values in a given data type into a valid index for the buckets array. This means that it is designed based on the size of the hash table's buckets array , the data type of the keys, and the range of valid values for keys in that data type.

Before to go any further, let's take an example. We want to implement a hash table which will hold pairs { K , V } where both K and V are strings representing words in different languages. To keep our proof of concept small, we will allow 20 buckets in this hash table.

Now we know that our hash function H() will take a string as parameter, length unknown, and compute an interval in the range [0..19]. There are many different ways we would be able to do so. What matters to us as far as qualities of this function?

- H() must associate a unique index to any valid key value

First are foremost, this means that H(k) needs to be deterministic; i.e. associates the same index H(K) to a given key K every time it is computed. That rules out having H(k) include any sort of random number since that'd mean that at 3PM your program would tell us that H("hello") = 3 and, at 5PM, that H("hello") = 17.

We also need H(k) to associate different indexes to different keys. For instance, if H("hello") is 3 and H("hi") is also 3, we would be unable to store pairs for both keys in our hash table. This rules out the idea of only using the first character of the string as a way to compute its key. We'll have to use all the characters, or, to keep the implementation simple, a sufficient subset. If our purpose is to store words in a language which will never exceed 20 characters in length, taking the 20 or maybe even 10 first characters to compute H(K) might be a reasonable assumption for implementing this specific hash table.

There is more though. What about H("team") and H("meat")? If you were thinking about simply adding the ASCII codes of each character with one another, you're in trouble. Any anagram key will produce the same hash value. We therefore need a formula which takes into account that a given character was at a given location in the string, i.e. its index in the string.

There are many solutions satisfying the above-mentioned constraints. The one we're going to use to illustrate the principles in our implementation is the polynomial hash code for strings.

A simple hash function

Given a string S composed of n characters S[0:n] = { s0, s2, …, s(n-1) } we first compute the ASCII code value for each of these characters and obtain X[0:n] = { x0, x1, …, x(n-1)}. We then define our polynomial hash coding function H1(S) as;

$$H1: string \rightarrow integers$$

$$H1(S) = H1(X) = \begin{cases} H1(\emptyset) = 0 \\ H1(X[a:n]) = xa + \ A * H1(X[a+1:n]) \end{cases}$$

A is a constant which value is 33, 37, 39 or 41. These have been found empirically to be best to reduce collisions, a concept we have not discussed yet but soon will.

Only remaining issue for now; the value we obtain isn't necessary in the [0..N-1] range. We therefore need a second function, referred to as a compression function, which will take the result of applying H1 to a key value and map it in the proper interval. While, again, the comparative performance of many approaches has been studied, we'll setting for a simple approach; using the modulus, aka remainder, operator.

$$H2: integers \rightarrow [0..N-1]$$
$$H2(v) = v \,\%\, N$$

The composition of both of these auxiliary functions will be used as our hash coding function;

$$H(S) = H2(\,H1(S))$$

While not optimal, this hash function will provide sufficiently good results for our proof of concept implementation. As soon as we're done revealing the mysteries behind the A constant, that is.

Collisions

Let's understand Collisions…

The last piece of the puzzle we have been sneakily working around so far is the concept of Collision. When we discussed simply summing the ASCII codes of all characters in a key string, we mentioned that we must avoid having all anagrams result in the same hash value. While hash function have been designed to avoid yielding such a result, e.g. polynomial sum vs. sum, the truth is that it will happen to have two distinct keys result in the same hash value. This is referred to as a collision and hash tables are designed so that they can handle such collisions.

If you want to store in a hash table about 20,000 entries, you might simply have 3 millions buckets so that even a mediocre hash function will seldom yield collisions. Not really memory efficient though. In practice, the buckets arrays are sized reasonably, given the number of entries you want to store, and hash functions are selected to have properties minimizing collisions. When a collision does occur, the GET, PUT and REMOVE operations are simply augmented with some code which will handle it.

…then ignore them…

Our simple implementation will not handle collisions, assuming that our buckets array an hash function are good enough to avoid them and simply refusing to add any pair { K1 , V1 } if H(K1) gives us an index corresponding to a bucket already containing a pair { K2 , V2 }.

But I don't wanna…

Fine. If you are still curious about how collisions should be handed, let's just mention one of the simplest ways to do so. Instead of storing a single pair {K , V} in each bucket, let's make each bucket the head of a linked list of such pairs. Ideally, each bucket will correspond to either an empty linked list or a linked list of a single element.

If a collision occurs during a PUT operation, it is simple to just add the new pair at the end or beginning of that linked list.

When we process a GET or REMOVE operation, we can't simply just assume that whatever is referenced by the bucket is the only pair. Other pairs might have collided with that same key. So now, we need to do more work, i.e. go through the linked list of bucket at index H(k) in the bucket array and verify whether our k matches any of the keys of the pairs stored in that linked list. If we find one, we can apply our GET or REMOTE operations to it. If we don't, then that key, even though it collides with the key of one or more pair in the hash table, is not associated with any element actually stored in the hash table.

Takes a little more work but, *on average*, given a good number of buckets and a good hash function, these operations remain in constant time.