Module 9 : HASH TABLES

Course Learning Outcomes:

1. Familiarize the use of Hash tables.
2. Learn on how to use a good hash Functions.
3. Learn the simple Hash Table in operation.

Hash Tables

In computing **Hash table** is a data structure that implements an associative array abstract data type, a structure that can map keys to values. A hash table uses a hash function to compute an index, also called a **hash code,** into an array of buckets or slots, which the desired value can be found.

The underlying idea of a hash table is very simple, and quite appealing: Assume that, given a key, there was a way of jumping straight to the entry for that key. Then we would never have to search at all, we could just go there! Of course, we still have to work out a way for that to be achieved. Assume that we have an array data to hold our entries. Now if we had a function h(k) that maps each key k to the index (an integer) where the associated entry will be stored, then we could just look up data[h(k)] to find the entry with the key k.

It would be easiest if we could just make the data array big enough to hold all the keys that might appear. For example, if we knew that the keys were the numbers from 0 to 99, then we could just create an array of size 100 and store the entry with key 67 in data[67], and so on. In this case, the function h would be the identity function h(k) = k. However, this idea is not very practical if we are dealing with a relatively small number of keys out of a huge collection of possible keys. For example, many American companies use their employees' 9-digit social security number as a key, even though they have nowhere near 109 employees. British National Insurance Numbers are even worse, because they are just as long and usually contain a mixture of letters and numbers. Clearly it would be very inefficient, if not impossible, to reserve space for all 109 social security numbers which might occur.

Instead, we use a non-trivial function h, the so-called hash function, to map the space of possible keys to the set of indices of our array. For example, if we had to store entries about 500 employees, we might create an array with 1000 entries and use three digits from their social security number (maybe the first or last three) to determine the place in the array where the records for each particular employee should be stored.

This approach sounds like a good idea, but there is a pretty obvious problem with it: What happens if two employees happen to have the same three digits? This is called a collision between the two keys. Much of the remainder of this chapter will be spent on the various strategies for dealing with such collisions.

First of all, of course, one should try to avoid collisions. If the keys that are likely to actually occur are not evenly spread throughout the space of all possible keys, particular attention should be paid to choosing the hash function h in such a way that collisions among them are less likely to occur. If, for example, the first three digits of a social security number had geographical meaning, then employees are particularly likely to have the three digits signifying the region where the company resides, and so choosing the first three digits as a hash function might result in many collisions. However, that problem might easily be avoided by a more prudent choice, such as the last three digits.

A simple Hash Table in operation

Let us assume that we have a small data array we wish to use, of size 11, and that our set of possible keys is the set of 3-character strings, where each character is in the range from A to Z. Obviously, this example is designed to illustrate the principle – typical real-world hash tables are usually very much bigger, involving arrays that may have a size of thousands, millions, or tens of millions, depending on the problem.

We now have to define a hash function which maps each string to an integer in the range 0 to 10. Let us consider one of the many possibilities. We first map each string to a number as follows: each character is mapped to an integer from 0 to 25 using its place in the alphabet (A is the first letter, so it goes to 0, B the second so it goes to 1, and so on, with Z getting value 25). The string X1X2X3 therefore gives us three numbers from 0 to 25, say k1, k2, and k3. We can then map the whole string to the number calculated as

$$k = k_1 * 26^2 + k_2 * 26^1 + k_3 * 26^0 = k_1 * 26^2 + k_2 * 26 + k_3.$$

That is, we think of the strings as coding numbers in base 26.

Now it is quite easy to go from any *number* k (rather than a string) to a number from 0 to 10. For example, we can take the remainder the number leaves when divided by 11. This is the *C* or *Java* modulus operation `k % 11`. So our hash function is

$$h(X_1X_2X_3) = (\mathbf{k_1} * 26^2 + \mathbf{k_2} * 26 + \mathbf{k_3})\%11 = \mathbf{k}\%11.$$

This *modulo* operation, and *modular arithmetic* more generally, are widely used when constructing good hash functions.

As a simple example of a hash table in operation, assume that we now wish to insert the following three-letter airport acronyms as keys (in this order) into our hash table: PHL, ORY, GCM, HKG, GLA, AKL, FRA, LAX, DCA. To make this easier, it is a good idea to start by listing the values the hash function takes for each of the keys:

Code	PHL	ORY	GCM	HKG	GLA	AKL	FRA	LAX	DCA
$h(X_1X_2X_3)$	4	8	6	4	8	7	5	1	1

It is clear already that we will have hash collisions to deal with.

We naturally start off with an empty table of the required size, i.e. 11:

Clearly we have to be able to tell whether a particular location in the array is still empty, or whether it has already been filled. We can assume that there is a *unique* key or entry (which is *never* associated with a record) which denotes that the position has not been filled yet. However, for clarity, this key will not appear in the pictures we use.

Now we can begin inserting the keys in order. The number associated with the first item PHL is 4, so we place it at index 4, giving:

				PHL						

Next is ORY, which gives us the number 8, so we get:

				PHL				ORY		

Then we have GCM, with value 6, giving:

				PHL		GCM		ORY		

Then HKG, which also has value 4, results in our first collision since the corresponding position has already been filled with PHL. Now we could, of course, try to deal with this by simply saying the table is full, but this gives such poor performance (due to the frequency with which collisions occur) that it is unacceptable.

Double Hashing

is a computer programming technique used in conjunction with open dressing in hash tables to resolve hash collisions, by using a secondary hash of the key as an offset when a collision occurs. Double hashing with open addressing is a classical data structure on a table.

The obvious way to avoid the clustering problems of linear probing is to do something slightly more sophisticated than trying every position to the left until we find an empty one. This is known as double hashing. We apply a secondary hash function to tell us how many slots to jump to look for an empty slot if a key's primary position has been filled already.

Like the primary hash function, there are many possible choices of the secondary hash function. In the above example, one thing we could do is take the same number k associated with the three-character code, and use the result of integer division by 11, instead of the remainder, as the secondary hash function. However, the resulting value might be bigger than 10, so to prevent the jump looping round back to, or beyond, the starting point, we first take the result of integer division by 11, and then take the remainder this result leaves when again divided by 11. Thus we would like to use as our secondary hash function h2(n) = (k/11)%11. However, this has yet another problem: it might give zero at some point, and we obviously cannot test 'every zeroth location'. An easy solution is to simply make the secondary hash function one if the above would evaluate to zero, that is:

$$h_2(n) = \begin{cases} \texttt{(k/11)\%11} & \text{if } \texttt{(k/11)\%11} \neq \texttt{0}, \\ 1 & \text{otherwise.} \end{cases}$$

The values of this for our example set of keys are given in the following table:

Code	PHL	ORY	GCM	HKG	GLA	AKL	FRA	LAX	DCA	BHX
$h_2(X_1X_2X_3)$	4	1	1	3	9	2	6	7	2	3

We can then proceed from the situation we were in when the first collision occurred:

				PHL		GCM		ORY		

with HKG the next key to insert, which gives a collision with PHL. Since $h_2(\text{HKG}) = 3$ we now try *every third location* to the left in order to find a free slot:

	HKG			PHL		GCM		ORY		

Note that this did not create a block. When we now try to insert GLA, we once again find its primary location blocked by ORY. Since $h_2(\text{GLA}) = 9$, we now try every ninth location. Counting to the left from ORY, that gets us (starting again from the back when we reach the first slot) to the last location overall:

	HKG			PHL		GCM		ORY		GLA

.

Note that we still have not got any blocks, which is good. Further note that most keys which share the same primary location with ORY and GLA will follow a different route when trying to find an empty slot, thus avoiding primary clustering. Here is the result when filling the table with the remaining keys given:

	HKG	DCA		PHL	FRA	GCM	AKL	ORY	LAX	GLA

Our example is too small to show convincingly that this method also avoids secondary clustering, but in general it does.

It is clear that the trivial secondary hash function $h_2(n) = 1$ reduces this approach to that of linear probing. It is also worth noting that, in both cases, proceeding to secondary positions *to the left* is merely a convention – it could equally well be *to the right* – but obviously it has to be made clear which direction has been chosen for a particular hash table.

Search complexity. The efficiency of double hashing is even more difficult to compute than that of linear probing, and therefore we shall just give the results without a derivation. With load factor λ, a successful search requires $(1/\lambda)\ln(1/(1-\lambda))$ comparisons on average, and an unsuccessful one requires $1/(1-\lambda)$. Note that it is the natural logarithm (to base $e = 2.71828\ldots$) that occurs here, rather than the usual logarithm to base 2. Thus, the hash table time complexity for search is again constant, i.e. $O(1)$.

Choosing good hash functions

In principle, any convenient function can be used as a primary hash function. However, what is important when choosing a *good* hash function is to make sure that it spreads the space of possible keys onto the set of hash table indices as evenly as possible, or more collisions than necessary will occur. Secondly, it is advantageous if any potential clusters in the space of possible keys are broken up (something that the remainder in a division will *not* do), because in that case we could end up with a 'continuous run' and associated clustering problems in the hash table. Therefore, when defining hash functions of strings of characters, it is never a good idea to make the last (or even the first) few characters decisive.

When choosing secondary hash functions, in order to avoid primary clustering, one has to make sure that different keys with the same primary position give *different* results when the secondary hash function is applied. Secondly, one has to be careful to ensure that the secondary hash function cannot result in a number which has a common divisor with the size of the hash table. For example, if the hash table has size 10, and we get a secondary hash function which gives 2 (or 4, 6 or 8) as a result, then only *half* of the locations will be checked, which might result in failure (an endless loop, for example) while the table is still half empty. Even for large hash tables, this can still be a problem if the secondary hash keys can be similarly large. A simple remedy for this is to always make the size of the hash table a prime number.

References and Supplementary Materials

Books and Journals

1. John Bullinaria ; March 2019 ; Data Structures and Algorithms ; Birmingham, UK;
2. Michael T Goodrich ; Roberto Tamassia ; David Mount ; Second Edition ; Data Structures and Algorithm in C++
3. www.google.com
4. Merriam Webster Dictionary