17 - hashing

A hash table is a quick way to access stored data; it is a frequently used and invaluable way to store date. It works by using the key itself as an instruction for finding the storage location.

In a hash table we have a **key**, a word in our case, and a **node** which stores the information we want to associate with the key along with a copy of the key itself. For definiteness we will imagine here that the key is a word. The idea behind a hash table is to store the nodes in an array, we will have to change this slightly, but imagine each node has an index, so, if the array was called `hash_array`, there would be nodes located at `hash_array[0]`, `hash_array[1]` and so on. The challenge is to go from the key to the correct index. The simplest strategy is to look in each node in turn, stopping when you find the correct one. This is clearly very slow, $O(n)$ in fact, if n is the number of nodes. Another strategy would be to make a big lookup table of all the indices and to use the alphabetic structure of the keys to search it using a binary search algorithm, this would be smart but it does rely on the keys having an ordering, they do here when the keys are words, but we want to solve the problem in a more general way.

The answer is to find some way of working out the index from the key itself, in other words, a hash table starts with a map, called the **hashing function** which maps each key to an index:

$$\begin{aligned} h : \text{keys} &\rightarrow \text{indices} && (1)\\ \text{key} &\mapsto h(\text{key}) && (2)\end{aligned}$$

What might this hashing function look like? Well that depends on the data, on the type of key, the performance constraints on the problem and the amount of data. An obvious, though poor, example for words would be to convert the letters into numbers with `a` going to 0, `b` to 1 and so on and then adding the values so

$$h(\text{`elbow'}) = 4 + 11 + 1 + 14 + 22 = 52 \tag{3}$$

Obviously this scheme would have to be changed slightly if there were capitals or other letters, so, for example, ascii coeds could be used.

The idea now is to have a big list `hash_table` and in this table `hash_table[52]` would store the node for 'elbow'. The crucial point is that the function always gives the same answer when applied to 'elbow', it is used to work out where to put the node associated with 'elbow' and later to access that node.

One immediate challenge is that we might have some large words, like zizzerzazzerzuzz, a creature in the Dr Seuss universe;

$$h(\text{`zizzerzazzerzuzz'}) = 295 \tag{4}$$

Obviously if this index was unexpectedly high and `hash_table` had less than 296 entries then this would be a problem. This problem could be solve by making sure the hash function had a predefined range, for example by using `mod` so

$$h(x) = \sum (\text{value of letters in } x) \% N \tag{5}$$

where N is the size of `hash_table`.

However, this discussion exposes a greater problem: 296 is not a big number compared to the number of words and if zizzerzazzerzuzz only has the hash value 295 there must be lots of

words that share the same hash value. Obviously two words that are anagrams of each other have the same hash value under this scheme:

$$h(\text{`male'}) = h(\text{`lame'}) \tag{6}$$

This situation can be improved by using a better hash function, the one I described above is straightforward but poor. We will examine better hash functions later. However, although the number of **collisions**, that is cases where

$$h(x_1) = h(x_2) \tag{7}$$

when $x_1 \neq x_2$, can be reduced by using a better clash functions, it is very hard to make sure it never happens; perfect hash function where there are no collisions are usually slow or impractical in other ways.

The answer to this problem is to make `hash_table` an array of linked lists instead of an array holding the nodes directly. Thus, when looking for the node with key x, you go to the linked list at `hash_table[`$h(x)$`]` and then search down the linked list until the node for key x is found. Of course, in fact, `hash_table` will not be in fact an array of linked lists, the items in an array must be of fixed size, so it will be an array of pointers to the heads of linked lists. This approach, called **direct chaining**, isn't the only way to deal with the problem of collisions, but it is one of the most straight forward.

Some hash functions

A choice of hash function is context dependent; typically a good hash function is one that avoids collisions but this depends on the data, different data sets will have different collision frequencies on the same data. They also depend on the size of the table, some hash functions map to specific numbers of indices. Density is also an issue: sometimes it is important that all the indices are used evenly. As such, there is no ideal hash function, it will always depend on the data and the application.

One popular hash function is Pearson hashing; it only produces 8-bit output, in other words, the hash table has to have size 256. It works using a look-up table mapping one `unsigned char` to another, in other words, it has a table that maps from one number in $\{0, 1, 2, \ldots, 255\}$ to another. Different look up tables will produce different hashing functions. It also relies on bitwise xor, bitwise xor often appears in hashing functions, mostly because it is the only binary bitwise operation that does not, on average, alter the number of ones and zeros. What Pearson's hashing does is it takes a starting 8-bit value and bitwise xor's it with the first letter; this gives a `unsigned char` which is then replaced by another `unsigned char` using the look-up table. This new value is bitwise xor'd with the second letter and all the steps follow as before. This carries out until the end of the word, with, each time, the letter being bitwise xor'd with the value calculated during the previous step. Example code is give in Table 1 and can also be found in the source folder.

A hash function that can map to any specified table length is given in the source folder as `djb2.c`.

```
1
2  unsigned char Pearson16(const unsigned char *x)
3  {
4
5    size_t i;
6
7    unsigned char h;
8
9    size_t len=strlen(x);
10
11   static const unsigned char lookup_table[256] = {
12
13     A LIST OF ALL THE NUMBERS IN 0-255 IN SOME ORDER
14
15       };
16
17   h = len; //or could initialize at, for example, zero
18
19   for (i = 0; i < len; ++i) {
20
21     h = lookup_table[h ^ x[i]];
22
23   }
24
25   return h;
26
27 }
```

Table 1: Pearson hashing. This print out doesn't include T, the lookup table.