Algorithms Worksheet 3 background - hash tables

The aim of this problem sheet is to implement a hash table and compare its performance to a linked list. A hash table is a quick way to access stored data; it is a frequently used and invaluable way to store date. It can be used in a very general way but as a definite example we will consider a hash table which counts how many times words are used in a book. This example was also used in the Haskell course where a trie was used. The clever thing about a trie is that the word itself was used as a kind of instruction for how to get to the location storing how many times the word was used. This is very elegant, but not as generic as a hash table. A hash table does share some part of the trick though, it uses the word as an instruction for finding the storage location.

In our problem we have a **key**, a word in our case, and a **bucket** which stores the information we want to associate with the key. In our case the bucket will hold the number of uses of the word. The idea behind a hash table is to store the buckets in an array, we will have to change this slightly, but imagine each bucket has an index, so, if the array was called `hash_array`, there would be buckets 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 bucket in turn and stopping when you find the correct one. This is clearly very slow, we will experiment with this in the exercise. Another strategy would be to make a big lookup table of all the indices and using the alphabetic structure of the keys to seach 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:

$$h : \text{keys} \rightarrow \text{indices} \tag{1}$$

$$\text{key} \mapsto h(\text{key}) \tag{2}$$

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 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 could be used.

The idea now is to have a big list `hash_table` and in this table `hash_table[52]` would store the bucket for 'elbow'. One immeadiate problem 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 have described is terrible. An example hash function is given in `djb2.c`. However, although the number of **collisions**, that is cases were

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

when $x_1 \neq x_2$, can be reduced, 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 buckets directly. Thus, when looking for the bucket with key x, you go to the linked list at `hash_table[`$h(x)$`]` and then search down the linked list until the bucket for key x is found. 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.