CI583: Data Structures and Operating Systems
Introduction to Hashtables

Hash tables

The hash table is an incredibly efficient data structure. Under the right conditions, it provides better selection and update performance than a BST or any type of self-balancing tree.

One of the only downsides of them is that they don't provide an easy way to visit the data in-order.

They were invented in the early 50s by a programmer called H.P. Luhn working for IBM, and independently by several others.

Applications include general purpose `key->value` maps in the standard library of every programming language, database indices, OS caching, and symbol tables used by compilers.

Hash tables

Values are stored in an array, to which keys provide an index. Cells in the array are often called buckets.

In order to translate a key (which might be a string, numeric or other value) into an array index, a hash function is used.

The hash function is a constant time function, as is array access, so the hash table provides $O(1)$ performance for insertion, deletion and search.

However, this is only true if the hash table is not more than about two-thirds full, for reasons that will be explained.

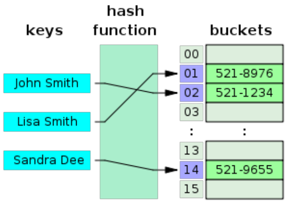


Image ©http://en.wikipedia.org/

background

Imagine that we needed to produce an employee database for a small company. Every employee has a unique payroll number and various other bits of information, such as their contact details, national insurance number, etc.

```
class Employee {
  int empNumber;
  String surname;
  String forename;
  //...
}
```

The employee numbers range from 1 to 1000 and there is no need for deletions – if an employee leaves, we want to keep their records on file.

Motivation

In this case it's perfectly reasonable to store `Employee` objects in an array and use employee numbers as keys.

Whenever we run out of space, we can create a new, larger array and copy the old records across. In this way we get the constant time performance of an array.

However, very few keys are as well-behaved as this one.

These keys run sequentially and there will be no deletions, meaning that we don't need to worry about fragmentation. The maximum employee number is a reasonable size for an array.

Motivation

What if we want to use a string as the key? This might be to store employee records by National Insurance number, or to store a dictionary of words and definitions.

Say we want to store the contents of a 50,000 word dictionary in an array. Each definition occupies its own cell and we can look it up without searching through the whole array if we know the index.

What we need is a way of converting a string into an appropriate index number.

Hash functions

We know that there are various encodings used to map characters to numbers, such as ASCII in which `a`=97, `b`=98 and so on.

However, ASCII runs from 0 to 255 and accommodates lots of non-printable characters and symbols. We can make a simpler system where `a`=1, `b`=2, up to `z`=26. We can make 0 stand for the space character, so we have 27 characters.

How can we use this to encode a sequence of characters?

Hash functions

A simplistic approach would be to sum the code numbers for each character. To encode the words “cats” we have

- `c`=3,
- `a`=1,
- `t`=20, and
- `s`=19.

Thus `cats`=43. If we restrict ourselves to 10-letter words the largest code is that for “zzzzzzzzzz”: 260.

Hash functions

So, we can only store 260 unique values, but our dictionary contains 50,000 word/definition pairs.

We could store a list of the definitions whose key have the same encoding in each cell – “tin”, “give”, “tend” and hundreds of other words all map to 43 in our encoding.

On average, each entry will contain a list of 192 definitions $(50,000/260)$ and we have lost the constant time performance.

Hash functions

Our array was too small before, so we need to make it bigger...

At the other end of the scale, we could store each definition in its own cell. So we need to map strings to unique indices.

Recall that we can think of a decimal number in terms of powers of 10. E.g.,

$$86,2486 = 8 \times 10^5 + 6 \times 10^4 + 2 \times 10^3 + 4 \times 10^2 + 8 \times 10^1 + 6 \times 10^0.$$

Hash functions

Similarly, we can break down a word into individual character codes and multiply them by powers of 27 (the radix of our encoding):

$$key(''cats'') = 3 \times 27^3 + 1 \times 27^2 + 20 \times 27^1 + 19 \times 27^0.$$

This does indeed give us a unique key for every string.

Hash functions

Unfortunately, the range of numbers has become too large. The biggest number, the key for “zzzzzzzzzz” is $26 \times 27^9 + 26 \times 27^8 \ldots 26 \times 27^0$ – well into the trillions. There are several reasons why we can’t handle an array that big!

The other problem is that most of the array will be empty – the scheme assigns an index to any combination of 10 characters, most of which aren’t English words.

Hash functions

We need to compress a huge range of numbers into one that will fit in a reasonably sized array.

We have 50,000 words to store, so you might presume we need an array with 50,000 elements, although we will actually want about twice that amount, as will be come clear.

We can do this using the `modulus` operator, %:

$$index = hugeNumber(key) \,\%\, arraySize.$$

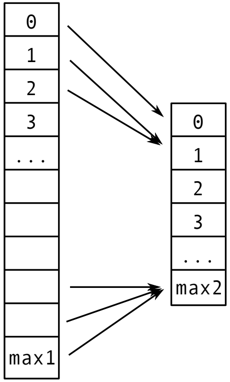
This is an example of a hash function. A perfect hash maps distinct keys to unique locations, but this is normally not possible.

Hash functions

In the case of our dictionary, `max1` is more than 7 trillion and `max2` is 100,000. Numbers in the big range will overflow numeric types, but we will deal with that later.

Several elements in the big range may map to a single element in the smaller range and some elements in the small range have nothing that maps to them.

On average, we have one entry for every two cells.



Handling collisions

Using a scheme like this, it is impossible to avoid collisions: this is the situation in which two words map to the same array index and this is the reason we make the hash table about twice as large as the data to be stored.

There are several ways we could respond when a collision occurs. The first, called open addressing is to search in some systematic way for an empty location and store the new value there.

The second approach would be to store a collection of entries at each index: this is called separate chaining, and makes it clear why each index in the array is sometimes called a bucket.

Open addressing

The simplest way to handle open addressing is called linear probing: If the location that we want to store a value in is occupied, we search sequentially for the next unoccupied location.

Say “cats” and “banana” have the same hash, 54321. If we have already stored the definition of “banana” and come to insert “cats”, we look at index 54322, then 54323, and so on, until we find somewhere to store the new value.

Open addressing

If we want to search for "cats" later on, we will first look at the index we get by hashing the key: 54321.

If it were unoccupied, then the search simply failed. In this case, the cell is occupied, but with the wrong data.

This should make it clear that we need to store the key as well as any other data.

E.g. if our hash table *just* maps words to their definitions, we would have looked up "cats" and retrieved the definition *A long curved fruit that grows in clusters and has soft pulpy flesh and yellow skin when ripe*.

Open addressing

So we are actually storing word/definition pairs and we can see that the value stored at 54321 isn't the right one.

We search forward sequentially – this is the linear probe. As soon as we come to an empty cell, we know the search has failed.

Open addressing

Searching like this makes deletions problematic: say “kumquat” also maps to 54321 and was inserted after “banana” but before “cats”, so that it occupies index 54322.

If we later delete “kumquat” we need to store a special value there, some sort of dummy `Nil` item.

Then the probe for “cats” does not stop when it encounters an empty cell.

The search for the location for a new entry can use one that contains this special value.

Open addressing

If there are a lot of deletions the table will contain lots of dummy `Nil` items, making searching less efficient. The number of items searched to retrieve or insert an item is called the probe length.

Many hash table implementations don't allow deletions for this reason. Similarly, duplicates are normally disallowed and inserting a value with an existing key just overwrites the existing value.

So, our insert algorithm will now probe for the first empty location, starting with the result of the hash function for the key, but will accept a location that already contains the key.