Hash tables

This section is a brief introduction to hash tables. We only introduce so-called *linked* collision management. The aim is just to present the concept of hash tables to get some understanding of their properties. In Python, dictionaries are implemented with hash methodology and there is hardly any reason to make your own implementations.

Introduction

Hash tables are an implementation of an abstract data type which stores items with unique *keys* with efficient support for the operations *search*, *insert* and *delete*.

Structur	Complexity for	
	search	*insert*
linked list	$\Theta(n)$	$\Theta(n)$
sorted array	$\Theta(\log n)$	$\Theta(n)$
binary search tree	$\Theta(\log n)$	$\Theta(\log n)$
hash table	$\Theta(1)$	$\Theta(1)$

In hash tables, you can thus search, insert and delete records in a constant time *no matter how many records are stored in the table*!

I can be proven that, on average, $\Omega(\log n)$ operations are required to locate a given key among n keys if you only allow keys to be compared by size. With that limitation one can say that binary search in arrays and search in (balanced) binary search trees are optimal — there is simply no way to do better.

However, it is possible to achieve faster methods if one, instead of basing search on *size comparisons*, uses the keys to *calculate the location*. The ideal would be that each key should have have an *unique* place (index in a list). However, since the amount of *possible* keys usually is far too large this is seldom achievable. That is when *hash tables* can be used.

Hash methodology

The records are stored in an indexable structure (an *array* or a *Python list*) called the *hash table*. The records are uniquely identified by a *key* (social security number, registration number, variable name, word, ...). In practice, the records often contain information other than the keys (e.g .address, owner, variable value, ...) but here we only look at the keys since they decides where the record is to be stored.

The hash table must have a fixed size and must be selected according to the *expected number* keys to be stored. For example, a register of Sweden's population should therefore have approximately $10,000,000,000$ places.

We also need a so-called *hash function* which specifies where in the table a certain key should be placed. The function should map the set of keys on the set of indexes. If the table size is m then the function should have a value in the range $[0 : m - 1]$.

There are two important questions: How should the hash function be constructed and how do we handle two different keys getting the same placement?

For now, we assume that the keys are positive integers. As hash function h we can then use the modulo function (the rest for integer division):

$$h(k) = k \bmod m$$

The modulo operation guarantees that the function value comes in the range $[0, m-1]$.

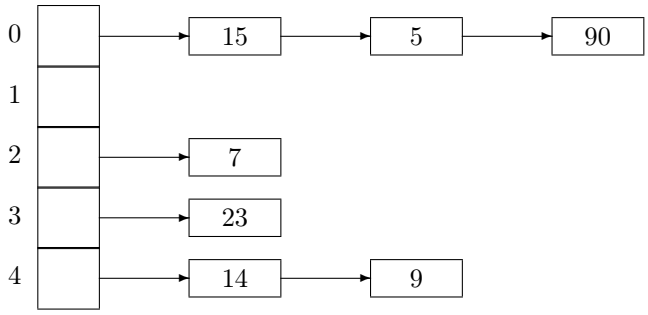
If two different keys k_1 and k_2 get the same location, this is called one *collision*. One way to handle this is to link together all elements that get the same location in a linked list. Usually the hash table T is then allowed to contain pointers to list elements. This is called *linked collision management*.

In e.g. Python you to use a regular list — it does not need to be linked.

Example

Assume that the keys 14, 15, 5, 7, 90, 23 and 9 are inserted in a hash table with 5 places ($m = 5$) and with linked collision management.

k	14	15	5	7	90	23	9
$h(k)$	4	0	0	2	0	3	4



Assuming that each key entered is equally likely to be searched, a successful search in this table requires

$$s = \frac{4 \cdot 1 + 2 \cdot 2 + 1 \cdot 3}{7} = \frac{11}{7} = 1.57 \tag{1}$$

on average.

A failed search requires on average

$$u = \frac{4 + 1 + 2 + 2 + 3}{5} = \frac{12}{5} = 2.4 \tag{2}$$

attempts if all keys are equally likely to be searched. (It takes one attempt to search an empty list.)

Analysis of linked collision management*

To analyze hash methods, we assume that a given key has the same probability of ending up in each of the m places.

If n keys are stored, the m lists are on average n/m long. To discover that a given key *does not* exist (a *failed* search) requires on the average

$$u(n, m) = 1 + n/m \tag{3}$$

attempts (searching an empty list requires, as we said, one attempt).

Searching for a key that exists (*successful* search) is a little harder to analyze because the average should be formed over the n keys – not over the number of table places.

We assume that each key is equally likely to be searched.

Observation: *It takes the same number of attempts to find an existing key as it was required to enter that key* which is the same as *the number of attempts required to search for that key before it was inserted.*

Thus, the average work of searching for an existing key is just as great as the work of building the table

$$\begin{aligned} s(n,m) &= \frac{1}{n}\left((1+\frac{0}{m})+(1+\frac{1}{m})+(1+\frac{2}{m})+\cdots+(1+\frac{n-1}{m})\right) \\ &= \frac{1}{n}\left(n+\frac{1}{m}\sum_{i=0}^{n-1} i\right) \\ &= 1+\frac{n(n-1)}{2mn} \\ &= 1+\frac{n-1}{2m} = 1+\frac{\alpha}{2}-\frac{1}{2m} \approx 1+\frac{\alpha}{2} \end{aligned} \tag{4}$$

where $\alpha = n/m$.

Hash functions

The function

$$h(k) = k \bmod m$$

which we have used in the example is not at all bad provided that m is a prime number. If m is not a prime number then the function becomes more sensitive to systematics in the keys.

A more general function is

$$h(k) = ((c_1 k + c_2) \bmod p) \bmod m$$

where $p > m$ is a large prime number, $p > 2^{20}$ (e.g. $p = 1048583$, c_1 a positive integer less than p and c_2 a non-negative integer less than p. The constant c_2 is often set to 0. Different c_1 provide completely different hash functions!

If the key is a string $s_1 s_2 s_3 \cdots s_k$ we can use the characters to produce a number e.g through

$$(((s_1 \cdot 37 + s_2) \cdot 37 + s_3) + \cdots) + s_k$$

and then use any of the above hash functions.

Hash tables in programming languages

Since the hash methodology is extremely fast, it is built into most modern programming languages and you need rarely implement this yourself.

In Python, *dictionaries*) is implemented as hash tables. Python allows a dictionary to grow. If the size of the hash table changes, the hash function must change, which means that already stored elements need to be reshashed and Python has advanced methods to do this automatically and efficiently.

Java has a number of classes which implement hash tables, among others a `HashMap` and a `HashSet`. Here it is the user's responsibility to provide an estimate of how many elements there are to be stored.