Hashing

The exposition is based on the following sources, which are all recommended reading:

1. Cormen, Leiserson, Rivest: Introduction to Algorithms, 1991, McGraw Hill (Chapter 12)
2. Sedgewick: Algorithmen in C++, 2002, Pearson, (Chapter 14)

Problem definition

Many applications require the support of operations INSERT, DELETE and SEARCH on a *dynamic set* which can grow and shrink over time.

Each element that can be inserted in the set has a *key* which is drawn from the *universe* U. The subset S that is stored in our set is comparatively very small, that is $|S| \ll |U|$.

What we would like is a data structure that supports the above operations if possible in time $O(1)$ while using only $O(|S|)$ space.

Problem definition (2)

Assume now that the subset S of U has size $n \ll |U|$. Obviously it takes too much space to allocate a table of size $|U|$.

Hence we simply allocate a table T of size $m = O(n)$ and *map* each element of S to a position in T. This is done by means of a *hash function* $h : U \mapsto \{0, 1, \dots, m-1\}$. A hash function should be computable in time $O(1)$.

Problem definition (3)

The obvious problem that occurs in hashing schemes is that of *collisions*, that is, the case that for two keys x and y with $x \neq y$ holds $h(x) = h(y)$.

There are several ways to deal with collisions. One is to avoid them altogether (which is possible using *perfect hashing*). If we allow for collisions there are two common methods to deal with them:

1. Chaining: We keep a linked list of the keys that hash to the same position in the hash table.
2. Open adressing: We store all keys in the table itself, and whenever a collision occurs, we use a secondary methods to locate another, free position in the table.

We will first talk about chaining and then about open adressing.

Hashing with chaining

The technique is straightforward and leads to the following running times for the basic operations:

1. SEARCH(T, k): This is proportional to the length of the list to which k hashes to. In the worst case $O(n)$.
2. INSERT(T, x): Here we have to append to a list in time $O(1)$.
3. DELETE(T, x): This is proportional to the length of the list to which k hashes to. Again in the worst case $O(n)$.

Hashing with chaining (2)

Of course the worst case scenario is not common. Usually the analysis is conducted under the assumption of *simple uniform hashing*, that means any element is equally likely to hash into any of the hash table slots, independently of the other elements. In that case we conduct the running time analysis in terms of the *load factor* $\alpha = n/m$.

It is not hard to show the following theorem:

Theorem.
In a hash table in which collisions are resolved by chaining, a (successful or unsuccessful) search takes time $\Theta(1+\alpha)$, on average, under the assumption of simple uniform hashing.

Hashing with chaining (3)

Proof.
To determine the expected number of elements examined during a successful search we take the average over all n elements in the table. More precisely, we take the average over 1 plus the expected length of the list when the i-th element is added. Under the assumption of simple uniform hashing this length is $\frac{i-1}{m}$, hence:

$$\begin{aligned}
\frac{1}{n}\sum_{i=1}^{n}(1+\frac{i-1}{m}) &= 1+\frac{1}{nm}\sum_{i=1}^{n}(i-1) && (1)\\
&= 1+\frac{1}{nm}(\frac{(n-1)n}{2}) && (2)\\
&= 1+\frac{\alpha}{2}-\frac{1}{2m} && (3)
\end{aligned}$$

Hence if $n = O(m)$, searching takes constant time on average. ■

Hashing with chaining (4)

In practice we choose—depending on the hash function—m a prime or a power of two (avoids modulo computations but has other disadvantages).

The problem with the above analysis is of course, that the assumption of simple uniform hashing does not always hold in reality. What can we do in that case? We then have to have a closer look at the hash function used.

Hash functions

A good hash function is of course crucial to the performance of any hashing scheme. It should come close to satisfying the assumption of simple uniform hashing, or more formally,

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \qquad j = 0,1,\ldots,m-1 \ ,$$

where $P(k)$ is the probability that the key k is drawn.

Unfortunately, P is generally unknown. We will now introduce some common classes of hash functions and assume for simplicity that the keys are natural numbers:

Hash functions (2)

In the *division method* we use $h(k) = k \mod m$. Here a few rules of thumb for choosing m.

1. Avoid powers of 2 as a value of m. Otherwise only the lowest order p bits will be used in the hash if $m = 2^p$.
2. Avoid powers of 10 when decimal numbers are used as keys.
3. Good values of p are primes not too close to a power of 2.

Hash functions (3)

In the *multiplication method* we use $h(k) = \lfloor m(kA \mod 1) \rfloor$.

1. The selection of m is not critical for this method. One can pick a power of 2.
2. The method works well with any A, but some values are better than others. According to Knuth a “good” value of A is $\frac{\sqrt{5}-1}{2} \approx 0.61803...$

Here an example of how the method works. Assume we have $k = 123456$, $m = 10000$ and $A = \frac{\sqrt{5}-1}{2}$, then

$$\begin{aligned} h(k) &= \lfloor 10000 \cdot (123456 \cdot \frac{\sqrt{5}-1}{2} \mod 1) \rfloor \\ &= \lfloor 10000 \cdot (76300.0041151 \mod 1) \rfloor \\ &= \lfloor 41.151 \rfloor \\ &= 41 \end{aligned}$$