Algorithmic Design / Algorithmic Data Mining

Data Mining - Lecture 3

Tommaso Padoan <tommaso.padoan@units.it>

Hash tables

Hash tables are another data structure for dynamic sets allowing the usual operations: search, insert and delete. As we will see, they are a generalisation of arrays. Actually, arrays themselves represent direct-access tables.

A *direct-access table* T associates every key in a set of natural numbers (universe) $U = \{0, \dots, m-1\}$, where m is the size of the table, with a slot or position. Every slot $T[k]$ can contain (a pointer to) a data item with the corresponding key k. Empty slots will simply contain *NIL* instead. The dynamic set operations are then as follows. Each one takes always constant $O(1)$ time.

```
def DirectSearch(T, k):
  return T[k]

def DirectInsert(T, z):
  T[z.key] = z

def DirectDelete(T, z):
  T[z.key] = None
```

Because of the total correspondence between positions indices and items keys, the universe U must be the same for both, which typically can be very large, hence take a lot of space. Moreover, the actual set K of used keys, corresponding to the slots actually occupied by an item, may be very small compared to the whole universe, thus most of the allocated space would be wasted. Another kind of problem is the fact that data items are required to have natural numbered keys, which .

Hash tables aim to solve those problems all together. They are still tables indexed by natural numbers in $\{0, \dots, m-1\}$, but indices are not required to come from the universe U of the keys, relaxing the previous requirements on the latter. A hash table uses a so-called *hash function* that maps keys to indices, defined as $h : U \to \{0, \dots, m-1\}$. Then, an item with key k will be stored into position $T[h(k)]$ of the table. In this way the space taken by the table can be reduced down to $\Theta(m)$, independently of the size of U. Furthermore, under some assumptions, the dynamic set operations still take constant $O(1)$ time, however this is true only for the average (and best) case, not anymore the worst case.

When the universe U still contains only integer keys, possible hash functions can be simply based on the modulo operation. For example, $h(k) = k \bmod m$, where m is the size of the table. When the universe contains other types of keys, such as strings, more complex hash functions may be adopted instead, although they should still be computable in constant time. In the following we will assume that that is the case.

Since table entries can be fewer than actual keys, two items, with different keys k_1 and k_2, may happen to be assigned the same slot, when $h(k_1) = h(k_2)$. This event is called *collision*, and must be resolved in some way, different from replacing the old item with the new one. A hash function may be better than another in evenly distributing the keys, but collisions are unavoidable as long as the table is strictly smaller than the number of possible keys, and may actually happen even if larger. The design of the search, insert and delete operations will depend on the choice of collision resolution method, possibly leading to different complexity.

Chaining

One of the simplest but effective ways to resolve collisions is by *chaining*. Chaining means to chain together items assigned to the same slot in a linked list, whose head is pointed by the corresponding table position. This may increase the cost of the dynamic set operations, because they may need to traverse the corresponding linked list before finding the item of interest. The three operations could be done as follows.

```
def HashSearch(T, k):
  return Search(T[h(k)], k)   # search key k in list T[h(k)]

def HashInsert(T, z):
  Insert(T[h(z.key)], z)      # insert z at the head of list T[h(z.key)]

def HashDelete(T, z):
  Delete(T[h(z.key)], z)      # delete z from list T[h(z.key)]
```

Recall that searching (and so deleting) in a linked list takes at most time linear in the length of the list, while inserting an item at the head of the list takes always constant time. Then, the table insert operation will always take constant $O(1)$ time. Instead, in the worst case when all contained and searched items have keys with the same hashing, the table search and delete operations can take up to linear time $O(n)$, where n is the number of items in the table. Obvisouly such a case is very rare, and will most likely never happen. Therefore the average-case complexity is worth analysing.

The *load factor* α of a table is the ratio n/m between the total number n of items currently contained in the table and its size m. The average-case performance strongly depends on how well the hash function distributes the set of keys among the m slots. For this analysis we will assume that **every key is equally likely to be hashed to any of the m slots, independently of the other keys, already encountered or not**. We call this the *simple uniform assumption.* This implies that every linked list $T[i]$, for $0 \leq i < m$, currently stored in the table, will have *expected length* $E[n_i] = n/m = \alpha$. Focusing on the search operation (deletion is similar) we can show that the average-case time complexity is $\Theta(1+\alpha)$. We separate the two cases when the searched key is actually contained in the table or not.

- If the searched key k is not in the table, then the whole linked list $T[h(k)]$ must be traversed. Thus, the expected number of items examined is the expected length α of the list, and the total time required, including that for computing $h(k)$, is $\Theta(1+\alpha)$.
- When the key k can be found in the table, the situation is a bit different because the probability that a list is searched is proportional to its length. The number of items examined before finding the one with key k is the number of items that come before it in the list $T[h(k)]$. Observe that those items must have been added after the searched item was inserted, since items are always added at the head of the list. By enumerating the n items contained in the table in the order in which they have been inserted, we obtain that the expected number of examined items when searching for the i-th item is $1 + \sum_{j=i+1}^{n} \frac{1}{m} = 1 + \frac{n-i}{m}$, since $\frac{1}{m}$ is the uniform probability for a key to be hashed to any specific index, in particular that of the searched item. Then, averaging over all n items we have:
$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{n-i}{m}\right) = \frac{1}{n}\left(\sum_{i=1}^{n}1+\sum_{i=1}^{n}\frac{n}{m}-\sum_{i=1}^{n}\frac{i}{m}\right) = 1+\frac{n}{m}-\frac{n+1}{2m} = 1+\frac{n-1}{2m} = \Theta(1+\alpha)$$

Since in both cases the time complexity is $\Theta(1+\alpha)$, independently from their actual probability, the average-case time complexity must be $\Theta(1+\alpha)$. Whenever $n = O(m)$, hence $\alpha = O(m)/m = O(1)$, search (and delete) are constant $O(1)$ time on average.

Exercise

Given a hash table with size 9, hash function $h(k) = k \bmod 9$ and collisions resolved by chaining.

1. Show what happens when inserting items with keys $5, 19, 15, 20, 33, 12, 17, 24, 10$ into the table, in the given order.
2. After inserting all items, what is the load factor of the table? What is the ratio of wasted space (positions containing *NIL*)?
3. How many items will be examined to search (unsuccessfully) the key 6?