Algorithmic Design / Algorithmic Data Mining

Data Mining - Lecture 4

Tommaso Padoan <tommaso.padoan@units.it>

Dynamic tables

A good dynamic set data structure should not only be able to search, insert and delete items, but also dynamically change its size depending on the number of items contained. This is indeed true, for instance, for red-black trees, but not for the (static) hash tables we saw.

A *dynamic table* is a (hash) table which resizes itself depending on the number of contained items. This is useful to both reduce the unused space and keep the load factor α under some desired value. The idea is to dynamically *expand* and *contract* the table whenever necessary, that is, during the insertion or deletion of an item. Some additional information must be recorded to decide when the table must be expanded or contracted. In particular, in a dynamic table T, the count of contained items is stored in an attribute $T.num$ and the current size of the table in $T.size$. Initially an empty table can have size 0, in which case we say that its load factor is 1. Otherwise, the load factor will be $\alpha = T.num/T.size$. To be precise, the actual table used for containing the items will also be an attribute $T.table$ of the dynamic table T.

Usually, it is desired to have a load factor $\alpha \leq 1$. This is required when the table can contain at most one item per slot, but it may be good also in other cases. Moreover, we wish the table to be filled for at least a constant factor of its size. Therefore, we expand the table, by doubling its size, only upon inserting a new item when the current load factor is already 1, that is when $T.num = T.size$. Similarly, we contract the table, by halving its size, upon deleting an item when the load factor would become less than some fixed value.

The procedures given below do not strictly depend on the kind of table used. They work for any table with as many slots as its size that allows for searching, inserting and deleting items. We generically call *Insert* and *Delete* the corresponding operations provided by the adopted table. Moreover, we call *CreateTable(m)* the procedure that creates (allocating the necessary space) a new empty table of size m. Note that, in the case of hash tables, the hash function depends on the size, thus, when creating a new one with different size, the hash function will also change. The search operation for the dynamic table would simply call the search operation provided by the underlying table.

Insertion

```
def DynamicInsert(T, z):
  if T.size == 0:
    T.table = CreateTable(1)
    T.size = 1
  if T.num == T.size:
    newT = CreateTable(2 * T.size)
    for x in T.table:
      Insert(newT, x)
    free(T.table)
    T.table = newT
    T.size = 2 * T.size
  Insert(T.table, z)
  T.num = T.num + 1
```

We analyse the running time of the dynamic insert operation by counting the number of elementary insertions performed by the procedure. Thus the results of the analysis should be multiplied by the actual time complexty of a single elementary insertion of the adopted kind of table, which in general is left open as a parameter. Note however that for hash tables we know that insertions take always constant $O(1)$ time, so it does not change the final dynamic table complexity.

When the table does not need to be expanded, the procedure performs just a single elementary insertion, thus costing $O(1)$ (insertions). On the other hand, whenever the table is expanded, the procedure also creates a new table and inserts all the current items into it, thus performing $T.num + 1$ elementary insertions, taking time linear in the current size of the table. So, if the table was initially empty, performing the n-th insertion can cost up to n elementary insertions, leading to $O(n)$ worst-case complexity. Observe that expansions do not depend on inputs, just on how full the table is. Therefore, averaging over all possible inputs would still lead to linear running time whenever the table is full, making an average-case analysis futile.

Amortized analysis

Amortized analysis is used to compute the average time required to perform an operation over a sequence of them. This kind of analysis is especially useful when, as in our case, an operation is usually very cheap but its cost may spike in specific situations. If such situations occur few enough times over every possible sequence, then every operation will still be cheap on average, because the spikes will be

amortized by the other very cheap occurrences. Such a cost is called the *amortized cost* of the operation. Note that probability is not involved, amortized analysis always guarantees the average cost of each operation to be at most the amortized cost, since some sequences may be globally cheaper than others.

Three different techniques can be used:

- *Aggregate analysis:* the amortized cost of an operation is obtained averaging its cost over any sequence of n operations simply by dividing the total cost of the sequence by its length n. When the sequence includes different operations, this approach will produce the same amortized cost for each one. For a more specific analysis of the various operations, one of the other two methods may be used instead.
- *Accounting method:* a different charge is assigned to each different operation in the sequence, which corresponds to its amortized cost. Whenever the assigned charge exceeds the actual cost of an operation, the difference is stored as credits. On the other hand, an higher cost can be paid using previous credits in addition to the current charge. Charges for each operation must be carefully chosen so that, for every possible sequence, the stored credits are always **non-negative** along the whole sequence.
- *Potential method:* a numeric value, called the potential, is assigned to each state of the data structure, using a fuction of the state. Thus, different potentials can be assigned to the same operation depending on the current state. Intuitively, the potential is a measure of the work already accounted for but not yet performed. The amortized cost of an operation is given by its actual cost plus the change in potential due to the operation. Similarly to the previous method, for every possible sequence, assigning a 0 potential to the initial state of the data structure, we want the potential to always be non-negative after every operation.

We will use simple aggregate analysis to show that the amortized cost of a single insert operation is constant $O(1)$, when considered in a sequence of n insertions. Enumerating the operations in the order in which they occur along the sequence, we know that the i-th one performs i elementary insertions if the table is expanded, that is, if $i-1$ is an exact power of 2, since after such insertion the table will contain i items. Otherwise, it performs a single elementary insertion. Then, the total cost of the sequence of n insertions is:

$$T(n) = n + \sum_{j=1}^{\lceil \log_2 n \rceil} 2^{j-1} < n + 2n = 3n$$

Thus, averaging over the n operations (dividing by n), we obtain a constant amortized cost of at most $3 = O(1)$.

Deletion

```
def DynamicDelete(T, z):
  Delete(T.table, z)
  T.num = T.num - 1
  if T.num == 0:
    free(T.table)
    T.table = None
    T.size = 0
  if T.num < T.size/4:
    newT = CreateTable(T.size / 2)
    for x in T.table:
      Insert(newT, x)
    free(T.table)
    T.table = newT
    T.size = T.size / 2
```

To ensure that the load factor is lower bounded by a positive constant, while preserving the constant amortized cost, when both insertions and deletions may occur, we require that contractions only happen, upon deleting an item, when α becomes less than $1/4$. We show that, indeed, the amortized cost of both insertions and deletions remains constant $O(1)$ using the accounting method, since it allows to assign different charges to the different operations (insert and delete) that can now happen along a sequence. Like before, we consider the actual cost of a dynamic deletion to be 1, for the elementary deletion of the item, plus the number of elementary insertions possibly performed when a contraction happens. Again, this cost is parameterised on the real costs of elementary insertions and deletions. For hash tables, the latter are constant time only in the avearage case, differently from insertions that require always constant time.

We apply the accounting method for amortized analysis as follows. The idea is that every time we perform an operation we will charge in advance for an extra that is supposed to cover part of the real cost of the next expansion or contraction. In particular, since expansions can only happen during insertions, while contractions in deletions, we will charge isnertions for the former and deletions for the latter. In particular, for every insertion we charge 3: 1 for the elementary insertion of the new item, 1 for the future re-insertion of that item during the next expansion, and 1 for the next future re-insertion of one of the items that were already in the table **before** the last expansion (of which there must be $T.size/2$ in total). For every contraction, instead, we charge 2: 1 for the actual deletion of the item and 1 for the future re-insertion of one of the remaining items. Note that, differently from insertions, we need not to invest for future operations on the deleted item, since it cannot be subject to expansions or contractions anymore. Moreover, the credits invested in the remaining items can ideally be moved to other remaining items if the former would be deleted before any contraction happened.

The charges defined above correspond to constant $O(1)$ amortized costs for both operations, 3 and 2 respectively. Then, we just need to prove that, along every possible sequence of n dynamic insert or delete operations, the credits never become negative. Note that, for each insertion of an item, at most one deletion of such item can occur, and it must occur after that insertion. For all $0 \leq i \leq n$, let

$size_i, num_i, \alpha_i, c_i$ be, respectively, the size of the table, the items contained, the load factor, and the stored credits after the first i operations in the sequence have been executed, where initially the table is empty, hence $size_0 = num_0 = 0$ and $\alpha_0 = 1$, and $c_0 = 0$ credits are stored. We can prove, by induction on i, that:

$c_i \geq 2 \cdot num_i - size_i$ if $\alpha_i \geq 1/2$, and

$c_i \geq size_i/2 - num_i$ if $\alpha_i < 1/2$.

Then, we can conclude that credits c_i are always non-negative, for all $0 \leq i \leq n$, because:

- if $\alpha_i \geq 1/2$, then $num_i \geq size_i/2$, and so $c_i \geq 2 \cdot num_i - size_i \geq 2 \cdot (size_i/2) - size_i = 0$
- if $\alpha_i < 1/2$, then $num_i < size_i/2$, and so $c_i \geq size_i/2 - num_i > size_i/2 - size_i/2 = 0$.

Finally, since the amortized cost of both kinds of operations is constant $O(1)$, we know that, along any sequence of n insertions or deletions, on average each operation takes constant time.

Exercise

Given a dynamic table T using a hash table with hash function of the form $h(k) = k \bmod T.size$ and collisions resolved by chaining.

1. Show what happens when inserting items with keys $5, 19, 15, 20, 33, 12$ into the table, in the given order.
2. After inserting all the items, show what happens when deleting the items with keys $5, 19, 20, 33, 12$ from the table, in the given order.
3. After all the previous operations, what is the size of the table? What is its load factor?