Eksamensnoter - Amortized Analysis

André Oskar Andersen (wpr684)

March 20, 2020

0.1 17 Amortized Analysis

- In an *amortized analysis*, we average the time required to perform a sequence of data-strcuture operations over all the operations performed. With amortized analyis, we can show that the average cost of an operation is small, if we average over a sequence of oeprations, even though a single operation within the sequence might be expensive.

17.1 Aggregate analysis

- In *aggregate analysis*, we show that for all n, a sequence of n operations takes *worst-case* time $T(n)$ in total. In worst case, the average cost, or *amortized cost*, per operation is therefore $T(n)/n$.

Stack operations

- In our first example of aggregate analysis, we analyze stacks that have been augmented with a new operation.
- The total cost of a sequence of n `PUSH` and `POP` operations is n, and the actual running time for n operations is therefore $\Theta(n)$
- Now wwe add the stack operation `MULTIPOP(S, k)`, which removes the k top objhects of stack S, popping the entire stack if the stack contains fewer than k objects. The total cost of `MULTIPOP` is $\min(s, k)$, and the actual running time is a linear function of this cost.
- Any seqyence of n `PUSH`, `POP` and `MULTIPOP` operations on an initially empty stack can cost at most $O(n)$, why?. We can pop each object fro mthe stack at most once for each time we have pushed it onto the stack. therefore, the number of times that `POP` can be called on an nonempty stack, including calls within `MULTIPOP`, is at most the number of `PUSH` operations, which is at most n. For any value of n, any sequence of n `PUSH`, `POP`, and `MULTIPOP` operations takes a total of $O(n)$time. The average cost of an operation is $O(n)/n = O(1)$. In aggregate analysis we assign the amortized cost of each operation to be the average cost. In this example, therefore, all three stack operations have an amortized cost of $O(1)$.

17.2 The accounting method

- In the *accounting method* of amortized analysis, we assign differing charges to different operations, with some operations charged more or less than they actually cost. We call the amount we charge an operation its *amortized cost*. When an operation's amortized cost exceeds its actual cost, we assign the difference to specific objects i nthe data structure as *credit*. Credit can help pay for later operations whose amortized cost is less than their actual cost.

- Different operations may have different amortized costs. This method differs fro maggregate analysis, in which all operations have the same amortized cost
- If we denote the actual cost of the ith operation by c_i and the amortized cost of the ith operation by $\hat{c}_i$, we require

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

  for all sequences of n operations.
- The total credit stored in the data structure is the difference between the total amortized cost and the total actual cost, or

$$\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$$

Stack operations

- To illustrate the accounting method for amortized analsis, let us return to the stack example. Recall that the actual costs of the operations were

`PUSH`	1
`POP`	1
`MULTIPOP`	$\min(k, s)$

- Let us assign the following amortized costs

`PUSH`	2
`POP`	0
`MULTIPOP`	0

- We shall now show that we can pay for any sequence of stack operations by charging the amortized costs. Suppose we use a dollar bill to represent each unit of cost. We start with an empty stack. When we push an object on the stack, we use 1 dollar to pay the actual cost of the push and are left with a credit of 1 dollar (out of the 2 dollars charged), which we leave on top of the plate. At any point in time, every plate on the stack has a dollar of credit on it
- The dollar stored on the object serves as prepayment for the cost of pop-pin it from the stack. When we execute a `POP` operation, we charge the operation nothing and pay its actual cost using the credit stored in the stack. To pop an object, we take the dollar of credit off the object and use it to pay the actual cost of the operation. Thus, by charging the `PUSH` operation a little bit more, we can charge the `POP` operation nothing

- Moreover, we can also charge `MULTIPOP` operations nothing. To pop the first object, we take the dollar of credit off the plate and use it to pay the actual cost of a `POP` operation. To pop a second plate, we again have a dollar of credit on the playe to pay for the `POP` operation, and so on. Thus, we have always charged enough up front to pay for `MULTIPOP` operations.
- Since each object on the stack has 1 dollar of credit on it, and the stack always has a nonnegative number of objects, we have ensured that the amount of credit is always nonnegative. Thus, for *any* sequence of n `PUSH`, `POP`, and `MULTIPOP` operations, the total amortized cost is an upper bound on the total actual cost. Since the total amortized cost is $O(n)$, so is the total actual cost

17.3 The potential method

- Instead of representing prepaid work as credit stored with specific objects in the data structure, the *potential method* of amortized analysis represents the prepaid work as "potential energy", or just "potential", which can be released to pay for future operations. We associate the potential with the data structure as a whole rather than with specific objects within the data structure
- The potential method works as follows:
  - We will perform n operations, starting with an initial data structure D_0.
  - For each $i = 1, 2, ..., n$ we let c_i be the actual cost of the ith operation and D_i be the data structure that results after applying the ith operation to data structure D_{i-1}.
  - A *potential function* Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the *potential* associated with data structure D_i.
  - The *amortized cost* $\hat{c_i}$ of the ith operation with respect to potential function Φ is defined by

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

  - The amortized cost of each operation is therefore its actual cost plus the change in potential due to the operation. The total amortized cost of the n operations is

$$\sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n} (c_1 + \Phi(D_i) - \Phi(D_{i-1})$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

– If we can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \hat{c_i}$ gives an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$

Stack operations

- To illustrate the potential method, we return once again to the example of the stack operations `PUSH`, `POP` and `MULTIPOP`.
- We define the potential function Φ on a stack to be the number of objects in the stack. For the empty stack D_0 with which we start, we have $\Phi(D_0) = 0$).
- Since the number of objects in the stack is never negative, the stack D_i that results after the ith operation has nonnegative potential, and thus

$$\Phi(D_i) \geq 0 = \Phi(D_0)$$

  The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.
- Let us now compute the amortized costs of the various stack operations.
- If the ith operation on a stack containing s objects is a `PUSH` operation, then the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1$$

  and the amortized cost of this `PUSH` operation is

$$\hat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$$

- Suppose that the ith operation on the stack is `MULTIPOP(S, k)`, which causes $k' = \min(k, s)$ objects to be popped of the stack. The actual cost of the operations is k', and the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'$$

  Thus, the amortized cost of the `MULTIPOP` operation is

$$\begin{aligned} \hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k' - k' \\ &= 0 \end{aligned}$$

  Similarly, the amortized cost of the `POP` operation is 0
- The amortized cost of each of the three operations is $O(1)$, and thus the total amortized cost of a sequence of n operations is $O(n)$. Since we have already argued that $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost. The worst-case cost of n operations is therefore $O(n)$

17.4 Dynamic tables

- In this section, we study the problem of dynamically expanding and contracting a table.
- We define the *load factor* $\alpha(T)$ of a nonempty table T to be the number of items stored in the table divided by the size of the table.
- We assign an empty table size 0, and we define its load factor to be 1.

17.4.1 Table expansion

- A table fills up when all slots have been used or, equivalently, when its load factor is 1.
- Upon inserting an item into a full table, we can *expand* the table by allocating a new table with more slots than the old table had. Because we always need the table to reside in contiguous memory, we must allocate a new array for the larger table and then copy items from the old table into the new table
- A common heuristic allocates a new table with trice as many slots as the old one. If the only table operations are insertions, then the load factor of the table is always at least 1/2, and thus the amount of wasted space never exceeds half the total space in the table
- The cost of the ith operations is

$$c_i = \begin{cases} i & \text{of } i-1 \text{ is an exact power of 2} \\ 1 & otherwise \end{cases}$$

The total cost of n `TABLE-INSERT` operations is therefore

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n \end{aligned}$$

because at most n operations cost 1 and the costs of the remaining operations form a geometric series. Since the total cost of n `TABLE-INSERT` operations is bounded by $3n$, the amortized cost of a single operation is at most 3

17.4.2 Table expansion and contraction

- When the number of items in the table drops too low, we allocate a new, smaller table and the ncopy the items fro mthe old table into the new one - we halve the table size when deleting an item causes the table to become less than 1/4 full. The oad factor of thetable is therefore bounded below by the constant 1/4