The University of Texas at Austin
Department of Computer Sciences
Professor Vijaya Ramachandran
CS357: ALGORITHMS, Spring 2006

Lecture 16

Amortized Analysis

1 Amortized Analysis

Given a data structure that supports certain operations, *amortized analysis* provides an upper bound on the average cost of each operation for any sequence of a given length n (i.e., an upper bound for a worst-case sequence).

By convention, amortized cost is specified as cost per operation, and not as cost of a sequence of operations of a given length.

Although we talk about the 'average cost per operation', *no probability is involved.*

While performing amortized analysis, we usually assume that the data structure starts from a canonical start configuration, which is typically that corresponding to the empty set.

Sometimes we analyze amortized cost per operation by type of operation.

There are three methods commonly used for amortized analysis:

- Aggregate method
- Accounting method
- Potential method

All three methods give the same solution (except that the solution returned by the aggregate method does not allow for different amortized costs for different types of operations), but they approach it in different ways.

Example. Consider a data structure MStack that supports the following 3 operations:

- Push(S, x): add element x to set S
- Pop(S): remove the most recently added element from S
- Multipop(S, k): let $l = \min(k, |S|)$; remove the l most recently added elements from S

We implement MStack as a standard stack. Then, Push and Pop take constant time. But the third operation takes $\Theta(l)$ time, since the Multipop operation takes l units of time to remove the l elements plus a constant time to access the stack and determine the value of l.

Question. *What is the amortized cost per operation of an MStack operation, assuming we start with the empty set S?*

1.1 Aggregate Analysis

In this type of analysis, we obtain an upper bound $T(n)$ on the time needed to execute any sequence of n operations, and hence derive the amortized cost per operation as $\leq T(n)/n$.

For the MStack we argue as follows:

Although a single Multipop operation can take as much as $\Theta(n)$ time in the worst case, over all Pop and Multipop operations, no more than $n-1$ elements can be removed from S since there are only n operations in total, and each Push operation can add exactly one element to S. Hence the total number of additions and removals of elements during these n operations is $< 2n$. Besides these addition/removal of elements we only spend constant time per operation. Hence for any sequence of n operations the total time needed to execute it is $O(n)$.

Hence amortized cost per operation is $O(n)/n$, which is a constant.

1.2 Accounting Method

In the accounting method, we assign an *amortized cost* $\hat{c}_i$ to each operation ahead of time, and we establish that these are valid amortized cost by showing that, for all sequences,

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

where c_i is the actual cost of the ith operation. The difference between the sum of the amortized costs and the actual costs is stored as credit on the elements in the data structure, and this credit is used to pay for future costly operations.

For the MStack, the actual costs of Push and Pop are 1 (i.e., a constant), and the cost of a Multipop is l.

We will use amortized costs of 2 for Push and 1 each for Pop and Multipop.

We now argue that the assigned amortized costs are valid. Each Push operation uses one unit of its amortized cost to execute the operation, and leaves the second unit as credit on the element placed on the stack. Thus each element in S resides on the stack with one unit of credit on it.

The Pop operation uses its one unit of amortized cost to test if the stack is empty or not. Similarly, the Multipop operation uses its one unit of amortized cost to compute to test the stack and compute the value of l. Other than this, each Pop and Multipop operation uses the credit on the element being removed to pay for the 'cost' of removing it. Thus, the sum of the amortized costs is always at least as large as the sum of the actual costs, (and the excess of the sum of the amortized costs over the actual sum is stored as credit on the elements in S).

Note that we used the value 1 to represent the constant cost of a Pop or Multipop operation outside of the actual deletions. This is a convention that is common in amortized analysis, where we ignore constant factors and simply use 1 (which represents the largest constant among the constants that may appear for different operations).

1.3 Potential Method

In the potential method we assign a potential Φ to the evolving data structure. Let D_i be the data structure after the ith operation, with D_0 the initial data structure.

The potential function we pick should satisfy $\Phi(D_i) \geq \Phi(D_0)$, for all $i > 0$. Usually, we pick a Φ with $\Phi(D_0) = 0$ and $\Phi(D_i) \geq 0$, for all i.

A potential function Φ that satisfies the above property *induces* an amortized cost $\hat{c}_i$ for the ith operation, for each $i > 0$, by

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

where c_i is the actual cost of the ith operation.

We observe that the induced amortized costs are valid since

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

For the MStack example, we use $\Phi(D_i) =$ number of elements on the stack. We can then observe the the amortized costs of Push and Pop remain constant since both the actual cost and the change in potential are constant for both operations, and the amortized cost of Multipop is also a constant since the drop in potential cancels out the actual cost of deleting multiple elements.

2 Re-sizing a Table

When we use a binary heap to implement a priority queue, we usually try to keep the *heapsize* to be linear in the number of elements n; of course we also need *heapsize* $\geq n$. However, we may not be able to stay with the table we started with due to Insert and Delete-min operations. A similar situation occurs when using a hash table to implement a dictionary; we will see this later.

We can achieve good *amortized* time per operation while maintaining linear space usage at all times by moving the items into an array (or 'table') of larger size when the load factor α gets large, and moving into a smaller table when α becomes too small. The parameters have to be chosen carefully to obtain constant amortized time per operation.

2.1 Table Expansion

Consider the case when only insertions are allowed (no deletions). In this situation we are looking for a strategy to move to tables of increasingly larger sizes so that the size of the table is always within a constant factor of the number of elements.

We use the following strategy for resizing the table:

- When the first element is inserted, create a table of size one and insert the element into that table.
- For each insertion after the first element, if the current table is full, i.e, $n > m$, then move all elements into a new table of size $2m$.

The time needed to move elements into the new table is proportional to the current number of elements n. We use the following potential function to show that the amortized cost per insertion is a constant.

$$\Phi(T) = 2n - m$$

2.2 Table Expansion and Contraction

If both insertions and deletions are allowed, we use the following strategy for resizing the table:

- If the table becomes full, i.e., $n > m$ then as in the table expansion case, we move the elements into a table of size $2m$.
- If table becomes less than *quarter full*, i.e., $4m < n$, then we move the items into a table of size $m/2$.

As before, the time needed to move elements into the new table is proportional to the current number of elements n. We use the following potential function to establish that the amortized cost of resizing the table is constant.

$$\Phi(T) = \begin{cases} 2n - m & \text{if } \alpha \geq 1/2 \\ (m/2) - n & \text{if } \alpha < 1/2 \end{cases}$$

By examining each of insertion and deletion for the case when table resizing occurs and the case when table resizing does not occur (4 cases in total) we can establish that the amortized cost per operation for resizing remains constant in all four cases. (Analysis is omitted.)