Notes on Amortization

D. Sleator

1. Introduction

A *data structure* is a way of representing information in a computer and a set of procedures for accessing and updating the information. These procedures for accessing and updating the information are called the *operations* on the data structure.

There are two fundamentally different ways in which data structures are used. They are used as part of an information retrieval system, and they are used as one component of an algorithm whose purpose is to solve some other problem.

Why do we have data structures? In the first application the purpose of the data structure is obvious, for the data structure itself is solving the problem that we want to solve. In the second application, the reason for having data structures is not so obvious. In this case our motivation for creating them is to aid in our conception of the algorithm. By using a data structure as a component of an algorithm, we split the problem of creating or expressing the algorithm into two parts. Once the interface between these two parts has been specified, these two components of the problem can be solved (or expressed) separately.

The interface between a data structure and the algorithm that uses it consists of two parts:

1. A set of operations that allow the algorithm to access and update the data structure.

2. Constraints that say how much time (or space) each operation is allowed to use in order for the algorithm to perform with the desired efficiency.

The structure of this interface implies that the data structure must perform in an on-line fashion, that is, it must perform the current operation before it knows what the future operations will be. Furthermore, no assumptions are made about the pattern of operations done by the algorithm. The data structure should have the desired performance for any sequence.

This note describes a technique that has been used to design improved data structures, and to analyze their performance. These advances were not made by changing the interface between the data structure and the algorithm that uses it. Rather, they were made by allowing the data structure to take full advantage of the flexibility of this interface.

The first observation is the following: Although the performance bounds on the data structure are specified by the interface, these bounds do not have to be satisfied by the data structure for every single operation. All that is actually needed is that the cost of sequence of operations be bounded by the sum of the specified bounds. An analysis of the worst case cost of a *sequence* of operations is called an *amortized analysis*.

For example suppose a sequence of operations $\sigma_1, \sigma_2, \cdots, \sigma_n$ is to be appled to a data structure. Say that the interface requires that the time taken by operation σ_i be at most $b(\sigma_i)$. In order for the data structure to satisfy the requirement of the interface, it is *not* necessary that:

$$t(\sigma_i) \leq b(\sigma_i),$$

rather, what is required is that

$$\sum_i^n t(\sigma_i) \leq \sum_i^n b(\sigma_i).$$

If the data structure has this property, then the operation σ_i is said to take *amortized time* $b(\sigma_i)$. (This definition of the amortized time of an operation is slightly more restricted than that used in the sequel. However, any bounds satisfying the above inequality certainly satisfy our more liberal definition given below.)

The second important observation is that althought the data structure cannot know the future operations, it is allowed to use the information it has about operations that were done in the past. It turns out that a useful technique in constructing data structures that are efficient in the amortized sense is to have the structure adjust itself based on past requests. Informally, we call such a data structure *self-adjusting*.

Amortized analysis and self-adjustment have been used to devise a variety of efficient new data structures. In the next section we illustrate the concepts of amortized analysis with a simple example.

2. Amortized analysis: an example

To illustrate the concepts of amortized analysis I shall use a simple example. Suppose that the cost of incrementing a binary number is the number of bits in it that change. What is the cost of incrementing a binary number from 0 to n?

It is easy to see that on each increment operation the low order bit changes. Thus, the number of times this bit changes is n. The 2's bit changes on the second increment operation, and on alternate subsequent operations, thus it changes a total of $\lfloor n/2 \rfloor \leq n/2$ times. Similarly, the ith bit changes at most $n/2^{i-1}$ times. Thus the total cost of this sequence of increments is at most

$$n + \frac{n}{2} + \frac{n}{4} + \cdots = 2n.$$

Thus, by the definition of amortized cost, the increment operation has an amortized cost of 2. Notice that some individual increment operation may cause $\lfloor \log n \rfloor$ bits to change. [1]

Another way to prove this result is my means of the *banker's view* of amortization. Suppose that each time a bit of the binary number changes it costs us one dollar. Also, suppose that we maintain a bank account such that for each bit of the binary

[1]The symbol "log" denotes the binary logarithm.

number that is a 1 we keep a dollar in the account. Initially there is no money in the account.

What happens when the binary number is incremented? A sequence of zero or more 1's all change to 0's, and then a 0 changes to a 1. For each bit that changes from a 1 to a 0, we take a dollar from the account to pay for it. For the bit that changes from a 0 to a 1, we must pay for changing the bit, and we must also put a dollar in the bank to maintain the relationship between the bank account and the number. Thus our out of pocket cost to pay for the increment is exactly two dollars, no matter how big the number is or how many carries occur. In a sequence of n increment operations, our total cost is $2n$ dollars and we are left with a non-negative bank account. Therefore the total cost of all the increments is at most $2n$.

This technique can be applied in a much more general way. The idea is to make a rule that says how much money must be kept in the bank as a function of the state of the data structure. Then a bound is obtained on how much money is required to pay for an operation and maintain the appropriate amount of money in the bank.

The *physicist's view* of amortization uses different terminology to describe the same idea. This is the formulation I shall use in this course. A *potential function* $\Phi(s)$ is a mapping from data state structure states to the reals. (This takes the place of the bank account in the banker's view.)

Consider a sequence of n operations $\sigma_1, \sigma_2, \ldots, \sigma_n$ the data structure. Let the sequence of states through which the data structure passes be $s_0, s_1, \ldots, s_n$. Notice that operation σ_i changes the state from s_{i-1} to s_i. Let the cost of operation σ_i be c_i. Define the amortized cost ac_i of operation σ_i by the following formula:

$$ac_i \;=\; c_i + \Phi(s_i) - \Phi(s_{i-1}), \tag{1}$$

or

$$(\text{amortized cost}) \;=\; (\text{actual cost}) + (\text{change in potential}).$$

If we sum both sides of this equation over all the operations, we obtain the following formula:

$$\sum_i ac_i \;=\; \sum_i (c_i + \Phi(s_i) - \Phi(s_{i-1}) \;=\; \Phi(s_n) - \Phi(s_0) + \sum_i c_i.$$

Rearranging we get

$$\sum_i c_i \;=\; (\sum_i ac_i) + \Phi(s_0) - \Phi(s_n). \tag{2}$$

If $\Phi(s_0) \leq \Phi(s_n)$ (as will frequently be the case) we get

$$\sum_i c_i \leq \sum_i ac_i. \tag{3}$$

Thus, if we can bound the amortized cost of each of the operations, and the final potential is at least as large as the initial potential, then the bound we obtained for the amortized cost applies to the actual cost.

We can now apply this technique to the problem of computing the cost of binary counting. Let the potential Φ be the number of 1's in the current number. Our first goal is to show that with this potential the amortized cost of an increment operation is 2.

Consider the ith increment operation that changes the number from $i-1$ to i. Let k be the number of carries that occur as a result of the increment. The cost of the operation is $k+1$. The change in potential caused by the operation is $-k+1$. (The number of bits that change from 1 to 0 is k and one bit changes from 0 to 1.) Therefore the amortized cost of the operation is

$$ac_i = k + 1 + (-k + 1) = 2.$$

Since the final potential is more than the initial potential, we can apply inequality (3) to obtain:

$$\sum_i c_i \leq \sum_i ac_i = 2n.$$

Notice that in this formulation, the definition of the amortized cost of an operation depends on the choice of the potential function. In fact, any choice of potential function whatsoever defines an amortized cost of each operation. However, these amortized bounds will not be useful unless $\Phi(s_0) - \Phi(s_n)$ is also bounded appropriately.

We have given two different definitions of amortized cost, one in Section 1, and the other in equation 1. Which definition applies in a discussion will depend on the context of the discussion. If we discuss amortized cost in the context of a potential function, then the amortized cost is that defined by equation 1. If it is outside the context of a potential function, then the meaning of amortized cost is that given in Section 1.

Most of the art of doing an amortized analysis is in choosing the right potential function. Once a potential function is chosen we must do two things:

1. Prove that with the chosen potential function, the amortized costs of the operations satisfy the desired bounds.

2. Bound the quantity $\Phi(s_0) - \Phi(s_n)$ appropriately.