CMSC 651 Advanced Algorithms	Lecture 3
Lecturer: Samir Khuller	Tu. Feb. 3, 1998

Original notes by Hsiwei Yu.

1 Overview of Course

Read [CLR] Chapter 18.1–18.3 for general amortization stuff.

1.1 Amortized Analysis

Typically, most data structures provide absolute guarantees on the worst case time for performing a single operation. We will study data structures that are unable to guarantee a good bound on the worst case time *per* operation, but will guarantee a good bound on the *average* time it takes to perform an operation. (For example, a sequence of m operations will be guaranteed to take $m \times T$ time, giving an average, or *amortized time* of T per operation. A single operation could take time more than T.)

Example 1: Consider a STACK with the following two operations: `Push(x)` pushes item x onto the stack, and `M-POP(k)` pop's the top-most k items from the stack (if they exist). Clearly, a single `M-POP` operation can take more than $O(1)$ time to execute, in fact the time is $\min(k, s)$ where s is the stack-size at that instant.

It should be evident, that a sequence of n operations however runs only in $O(n)$ time, yielding an "average" time of $O(1)$ per operation. (Each item that is pushed into the stack can be popped at most once.)

There are fairly simple formal *schemes* that formalize this very argument. The first one is called the **accounting method**. We shall now assume that our computer is like a vending machine. We can put in \$ 1 into the machine, and make it run for a *constant* number of steps (we can pick the constant). Each time we push an item onto the stack we use \$ 2 in doing this operation. We spend \$ 1 in performing the push operation, and the other \$ 1 is stored *with* the item on the stack. (This is only for analyzing the algorithm, the actual algorithm does not have to keep track of this money.) When we execute a multiple pop operation, the work done for each pop is paid for by the money stored with the item itself.

The second scheme is the **potential method**. We define the potential Φ for a data structure D. The potential maps the current "state" of the data structure to a real number, based on its current configuration.

In a sequence of operations, the data structure transforms itself from state D_{i-1} to D_i (starting at D_0). The *real cost* of this transition is c_i (for changing the data structure). The potential function satisfies the following properties:

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

We define the *amortized cost* to be $c'_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$, where c_i is the true cost for the i^{th} operation.

Clearly,

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

Thus, if the potential function is always positive and $\Phi(D_0) = 0$, then the amortized cost is an upper bound on the real cost. Notice that even though the cost of each individual operation may not be constant, we may be able to show that the cost over any sequence of length n is $O(n)$. (In most applications, where data structures are used as a part of an algorithm; we need to use the data structure for over a sequence of operations and hence analyzing the data structure's performance over a sequence of operations is a very reasonable thing to do.)

In the stack example, we can define the potential to be the number of items on the stack. (Exercise: work out the amortized costs for each operation to see that Push has an amortized cost of 2, and M-Pop has an amortized cost of 1.)

The rest of the lecture was spent finishing the proof from last time.