Second Level Algorithms

Prof. Palash Dey

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Week – 01

Lecture 04

Welcome to the fourth lecture of the second-level algorithms course. We have been studying amortized analysis and have seen two methods, namely aggregate analysis and accounting methods, for analyzing the amortized time complexity of various algorithms. In this lecture, we will see a third method, which is the potential method, for analyzing the amortized time complexity of various algorithms. So today's topic is the potential method. So in this method, we define something called the potential of every state of the data structure.

The potential is always non-negative. That is how it is $\hat{c}_i$, if we denote by $\hat{c}_i$ the amortized cost of the i-th operation and π the actual cost of the i-th operation, then we need to satisfy the following: the amortized cost taking an analogy from the accounting method, the amortized cost is the cost that we charge to the i-th operation, and that should be equal to c_i, the actual cost, plus Think of potential as the total amount of money stored in the data structure, with an analogy to the accounting method. So, in the accounting method, the money was distributed across various places in the data structure. In the potential method, you can think of the total money as stored in a bank, and that amount is called potential. So, if D_i is the state of the data structure after the i-th operation and D_{i-1} was the state of the data structure after the (i-1)-th operation, the change in bank balance is $\Phi(D_i)-\Phi(D_{i-1})$.

So, this much should be equal to $\hat{c}_i$. So, if this is the case, if this is so, where D_i and D_{i-1} are respectively the states of the data structure after the i-th and (i-1)-th operation, and $\Phi(D_i)-\Phi(D_{i-1})$ are the corresponding potentials. This is the case.

Now, consider a sequence of n operations. and this equality holds that $\hat{c}_i$ is $c_i+\Phi(D_i)-\Phi(D_{i-1})$. If this holds for all i, then the total amortized cost is the total cost this is the total cost, and this sum telescopes, so this sum is this. Now, if we can define

the potential in such a way that the initial potential is 0. Then we have total amortized cost is total actual cost plus the final potential. Now, because potentials are non-negative, we have total amortized cost is at least total actual cost. which we need to prove, which we need to claim that the amortized, the claimed amortized cost $\hat{c}_i$ is actually the upper bound of the amortized cost. Again, as with the accounting method, this seems a bit abstract.

So again, let us see examples. So our first example, again is multi stack push x which takes $O(1)$ time in the worst case. pop which takes $O(1)$ time in the worst case and multipop k which takes $O(k)$times in the worst case. All three operations we have already seen that they take $O(1)$ amortized time.

So, we will prove this again using potential method. So, in the potential method all we need to prove is this equality that the amortized cost is actual cost plus the change in potential. Once we prove this then the rest From this it follows that the total amortized cost is greater than equal to total actual cost. So in the potential function the main non-triviality is to define the potential.

So the way we define it here is we define the potential of the stack recall potential is a function from the states of the algorithm or data structure in this case a stack. So, for every state of the stack we have to define its potential. So, this is how we define it.

We define the potential of the stack to be the number of elements. It is currently storing. So, you see by definition the potential is always non-negative because the number of elements that the stack is storing cannot be negative. The potential is always non-negative, okay? And initially, the stack is empty, so phi of the initial state of the algorithm or data structure is 0.

So, now we have to define the $\hat{c}_i$ function for push, pop, and multipop. So, $\hat{c}_i$ is we define it to be 2 if the i-th operation is a push, 0 if it is a pop, and 0 if it is a multipop. So, let us prove that the equality holds, that is, $\hat{c}_i$ to prove.

That the potential, the amortized cost, is the actual cost plus the change in potential. So, three cases because there are three operations. The i-th operation is a push operation, then LHS is $\hat{c}_i$, which is 2. RHS is the actual cost plus the change in potential. The actual cost is 1. So, all costs of $O(1)$ we will write as 1.

and the potential of the data structure drops by 1 that means this change in potential is sorry the potential increases by 1 because it is a push operation. It does not drop. It

increases by 1. Recall the potential of the data structure is the number of elements it is storing.

It is a push operation. So, the potential increases by 1. The change in potential is also 1. So, the RHS is also 2. So, when I is a push operation, this equality hold.

So, let us see for the pop operation. If the i-th operation is a pop operation then LHS which is the amortized cost, the amortized cost of pop operation we are claiming it to be 0 and RHS is actual cost plus change in potential. So, actual cost is 1, but the potential drops by 1 because it is a pop operation. Number of elements in the stack drops by 1. So, change in potential is -1.

So, this is also 0. So, if the ith operation is a pop operation, then also LHS equals RHS. If the i-th operation is a multipop operation on a stack storing S elements, then LHS is the amortized cost, which we claim to be zero; RHS is the actual cost plus the change in potential.

So, the actual cost is the minimum of K and S, and here also it is multipop. So, the potential drops by the number of elements popped, which is minus. The number of elements popped, hence the array trace is also 0. So, for all three operations, this equality holds, and hence the amortized cost of push is 2, and pop and multipop is 0, which can be written as the amortized cost of all three operations is O. So, next, let us see our second example.

A k-bit binary counter, which takes big O of k time in the worst case, and we have already seen that the amortized time complexity of increment is O. We will reprove this using the potential method. So, as usual, the most interesting and non-trivial step is defining the potential. So, here we define the potential.

of the counter to be the number of 1 bit in it. Hence, the potential is always non negative. and if initially the binary counter was 0, then P of d 0 is 0 good. Now, we have to show that the amortized cost is the actual cost plus change in potential.

So, suppose the i-th increment operation changes l bits from 1 to 0. So, it can change many bits from 1 to 0, but as we have already at most one bit can change from 0 to 1. So, we charge the i-th operation $\hat{c}_i$ the amortized

the number of bits changed from 0 to 1 which we know is at most 1 Okay, so now let us try to prove LHS. So we have to prove that amortized cost is equal to actual cost plus

change in potential. So LHS is the amortized cost which is the number of bits changed from 0 to 1 and RHS is the actual cost plus change in potential

actual cost is the number of bits changed from 0 to 1 plus the number of bits changed from 1 to 0 ok. And what is changed in potential? Then it drops by the number of bits that are changed from 1 to 0 and it increases by the number of bits that are changed from 0 to 1. So, this is minus the number of bits

changed from 1 to 0 plus the number of bits changed from 0 to 1. So, this is 2 times the number of bits changed from 0 to 1. LHS is the number of bits changed from 0 to 1. So, instead of charging that as amortized cost, we let us charge 2 times it.

So, that LHS is same as RHS. the amortized cost is 2 times the number of bits changed from 0 to 1, which number is 1 and hence the amortized cost is at most 2. So, LHS is 2 times the number of bits changed from 0 to 1 and hence we have LHS equal to RHS. So, this proves that The amortized cost of incrementing k-bit counter

It is $O(1)$. So, these are the three methods, and here is something interesting: if we add some operation, then the amortized cost can change. So, here is an interesting claim. That if we allow both increment, of course modulo 2^k, and decrement modulo 2^k, then the amortized cost

of both increment and decrement is $O(k)$, OK, which is the same as their worst-case cost. How do you prove it? So, to prove it, it is enough to give a sequence of n operations for arbitrarily large n which takes $O(kn)$ value. How do you do it?

So, here is a sequence of n operations. So, first you increment 2^k-1 times. So, first 2^k-1 times—this many times you increment—and after that, the state of the binary counter will be all 1. Then you toggle between increment and decrement. Increment, decrement.

this thing you continue $\frac{n}{2}-(2^k-1)$ times. So, now total actual is you see the first 2^k-1 increment operations the total amortized cost is $O(2^k)$ but after that every increment and decrement each one takes k bits so it toggles all the bits so the rest one takes $O(k)$ times each operation takes $O(k)$ times and the number of operations is $n-2^k+1$ okay so this is so not only big O it is actually omega let us write omega because we are doing lower bound So, this is, you know, $\Omega(nk)$ for n much larger than k, okay.

Hence, the amortized time complexity of increment and decrement is $\Omega(k)$. Okay, so in the next class we will see a very interesting data structure which is called dynamic tables

and we will show that the amortized time complexity of various operations in dynamic table is $O(1)$. Although the worst case time complexity could be much higher for the operations of dynamic table. Okay, so let us stop here.

Thank you.