Second Level Algorithms

Prof. Palash Dey

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Week – 01

Lecture 03

Welcome to the third lecture of the second-level algorithms course. So in this lecture, we will continue to study aggregate analysis for the amortized aggregate method for amortized analysis. So let us begin. Aggregate method for amortized analysis. Okay, so in the last lecture, we have seen the example of multistack, and we have shown that all three operations of multistack take big O of one amortized time. Then we started studying the binary counter and the increment function, and we have seen that incrementing—that is, adding 1 modulo 2^k, where k is the number of bits in the binary counter—takes $O(k)$ time in the worst case. So, in today's lecture, we will see that the increment function takes $O(1)$ amortized time. So, to get a feel, let us write down the counter value—the initial first values—so here is $b_{k-1} \cdots b_3 b_2 b_1 b_0$. So we assume that initially, the counter value is 0, so all bits are 0 right, and then if I increment, it becomes 0, this 1. Again, if I further increment, this is what I get and so on. So, a crucial thing to observe is that if we consider any sequence of n increments on a k-bit binary counter which is initialized at 0. Let us ask how many times the least significant bit b_0 changes. You see, in every increment, b_0 changes. So here you see 0 to 1, then 1 to 0, 1 to 0 ok. So, the bit b_0 changes n times every time I increment the binary counter b_0 toggles. let us ask how many times b_1 changes now b_1 changes only one alternate times so if I perform n many operations then the bit b_1 changes floor of $\frac{n}{2}$ times. Let us ask how many times the bit b_2 changes.

Here you see b_2 changes once every 4 operations. So, the bit b_2 changes $\frac{n}{4}$ floor of it times. In general the bit b_i changes floor of $\frac{n}{2^i}$ times ok.

So, what is the total number of change of bits in this sequence of n increment operations? So, it is sum over i equal to 0 to k number of times bit b_i changes which is floor of $\frac{n}{2^i}$ this is less than equal to $\sum \frac{n}{2^i} \leq \sum \frac{n}{2^i} \leq 2n$. So, the total number of changes in the bits in this sequence of n increments is at most twice n. So, the amortized time complexity of increment which is proportional to the number of bit changes is $O\left(\frac{2n}{n}\right)$ which is $O(1)$.

So, this concludes our proof that the amortized time complexity of increment is $O(1)$. So, now let us see another method for finding the amortized time complexity of various algorithms. The second method, which is called the accounting method. In this method, we charge every operation its claimed amortized cost. Okay, so charge some amount of money—we assume that one rupee can pay for any $O(1)$ time operation. We need to show that for any sequence of k operations, the total amortized cost—which is the total amount charged, that is, the total amount charged—is at least the total actual cost of this sequence of k operations, okay.

So, in the accounting method, we will store the extra amount that we have in the data structure itself, and we will show that our balance—the total amount stored in the data structure—is never negative, thereby automatically proving that the total amount charged is at least the total cost of this sequence of k operations. It may seem a bit abstract now. So, let us see how this method is used with those two examples that we have seen: the multistack and the binary counter increment. So, the first example is Example 1: multistack.

Recall it has 3 operations: push, which pushes a new key on top of the stack; pop; and multipop. Multipop k pops the minimum of k and the number of elements in the stack. So, if the stack has at least k elements, then it pops k elements. So, the worst-case time complexity of push is $O(1)$, pop is $O(1)$, and multipop is $O(k)$. And we have already seen, using the aggregate method, that the amortized time complexity of all these three methods is big O of 1. Now, let us reprove that using the accounting method.

So, what we do is we charge each push operation 2 rupees, okay. Now, push operation takes $O(1)$ time, and we can every $O(1)$ time operation can be paid off using 1 rupee. So, we pay 1 rupee for the $O(1)$ time push operation.

Okay, and so we have one rupee left, which we store in the data structure with that element pushed. So, it is not, you know, we actually store it; it is just for analysis. The remaining One rupee is stored with the newly pushed element. Okay, so now when pop happens, we do not charge any amount for pop, okay. But pop takes $O(1)$ time. So, the $O(1)$ time cost of the pop operation is paid by the rupees one amount stored with the element being popped. you see because of the charging of the push operation every element in the stack has one unit of one unit of money stored with it which will be used for popping that element in the future pop operation similarly we do not charge any amount for multipop the $O(k)$ time actual cost of multipop is paid by the rupees key amount stored with the top key element being popped.

So, this way every operation is fully paid. And this proves that the amortized cost of push, pop and multipop is O . Next, let us go to our next second example, which is incrementing k bit binary counter. So, here also this increment operation takes bigger of k time in the worst case, but we have already seen using aggregate method that the amortized time complexity of increment is bigger of 1. So, now let us prove that again using accounting method.

So, in the increment a crucial observation is that in every increment exactly one bit changes from 0 to 1 in every increment, although many bits can change from 1 to 0, exactly one bit changes from 0 to 1, except all 1 to all 0 base 2.

In this case, no bit changes from 0 to 1, so even better. So, what we do? We charge 2 rupees for every increment. We pay 1 rupee to toggle any bit to change any bit from 0 to 1 which is enough because in every increment at most one bit changes from 0 to 1 and we store the remaining one rupee with the bit that has been changed from 0 to 1 in this increment ok. So, all the bits all the 1 bits in the data structure store 1 rupee. So, all

the 1 bit store rupee 1 which is used to change in future. them to 0. So, all the 1 bits store the amount of money which is needed to change them from 1 to 0. So, we only pay for 0 to 1 bit and then we charge 2 rupees.

So, an extra 1 rupee is stored with the newly made 1 bit. So, this shows that the amortized time complexity of increment is O(1). So, let us stop here. Thank you.