Second Level Algorithms

Prof. Palash Dey

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Week – 02

Lecture 06

Welcome to the sixth lecture of the second-level algorithms course. So, in the last class, we have seen the dynamic table data structure, and we have seen the insertion process. Although the worst-case time complexity of insertion is $O(n)$, we have seen that the amortized time complexity of insertion is $O(1)$, and we have used aggregate analysis to analyze the amortized time complexity of the insertion process. Then we have seen the naive deletion, which simply deletes the key without resizing the array, and with that insertion and deletion, the amortized time complexity of both operations is $O(1)$. But then, that naive deletion gives rise to the low load factor of the table, resulting in inefficient use of space.

So, we have seen another delete-key operation, which resizes the table when the load factor of the table drops below half. And then we ended the lecture with a claim, which I gave as homework to disprove—that the amortized time complexity of insert and modified delete is $O(1)$. That is the wrong statement, and you have to prove or disprove it. So, in this lecture, we will modify the deletion procedure to achieve $O(1)$ amortized time complexity for both insertion and deletion simultaneously, okay? So, let us begin. Dynamic Table Continuation.

So, the modified deletion procedure. So, we will make a simple change: instead of resizing the table or array when the load factor drops below half, we will resize the array when the load factor drops below 1 by the load factor of the table drops below $\frac{1}{4}$ after deleting any key. Then we create another array of size half the size of the current array and copy all the elements.

of the old array to the new array notice that when we resize the table in ah delete procedure, the load factor of the new table becomes half. because the load factor was $\frac{1}{4}$

or less it fell just below $\frac{1}{4}$. That means, the table was $\frac{1}{4}$th full now the array size becomes half.

So, with respect to the new array the table is now half full. So, this is in contrast with the table resize during insert procedure which makes the new load factor 1 ok, during new load factor again half. So, it is not actually not in contrast it is actually this is the same.

So, in the insert procedure also when the load factor is 1 and another insert happens then we increase the table size make the table size double and the load factor becomes half. So, it is in contrast with the old deletion process. So, in the old deletion process which fail to get give us a $O(1)$ amortized time complexity for insert and delete. There we are making the table size half when the load factor drops just by just below half. So, in the old deletion process.

The load factor of the newly created table after resizing during deletion becomes 1, and that is the main problem why we could not achieve the $O(1)$ time amortized time complexity for insertion and deletion. But in this case, that is not the problem, and actually, we will now claim that the amortized time complexity of insert and delete procedures as described above is $O(1)$. Now, when there are multiple different procedures involved.

So, what do we need to show? We need to show proof. We need to show that any sequence of n operations where each operation is an insert or delete takes a total of $O(n)$ time.

So, I take any arbitrary sequence of n operations where each operation is an insert or delete. So, there are 2^n such sequences of length n, and we have to show that for all such sequences, the total time required to process all n operations is $O(n)$. So, typically when multiple operations are involved, the use of aggregate analysis to bound the amortized time complexity of the operations becomes challenging, and typically the accounting method or potential method becomes useful. So, we will see a proof of this claim using the accounting method. We will prove this.

claim using accounting method ok. So, recall in accounting method money is distributed across the data structure. So, we will maintain this invariant. So, invariant whenever we resize the table during insertion or deletion.

we will ensure that every element of the table has at least rupees 1 stored with it. Hence, during resize every element in the old table pays for its movement from the old table to the new table.

So, this species will ensure will charge accordingly. So, whenever we are resizing every element in that table which are we are currently present have 1 rupee with them which will which will be used for copying them from the from the old table to the new table copying each element from the old table to new table takes $O(1)$ time and recall we have assumed that using 1 rupee we can pay for any operation which takes $O(1)$ time. So, now, let us move on and see how much we charge each operation. We charge every insert operation rupees 2 and every delete operation rupees 2 ok. So, let us see whether this charging is enough or not and if it is not enough we will increase it. So, how does the insert operation takes place? I will recall if there is a space in the table then.

So, suppose the table looks like this and this is the count this is the size by 2 and suppose this is the count value current size of the table and. So, when a new entry new key comes for insertion will store that key in this particular index and what we and this particular operation takes $O(1)$ time ok. So, if insertion does not result in resizing the table, then its actual cost is $O(1)$ actual time complexity. We use 1 rupee to pay for this actual cost ok. But you see we are keep on inserting and when the table is full I want the guarantee the invariant says that every element in the table must store 1 rupee. So, we must store this money somewhere in the table.

So, a natural thing is that you know you have charged 2 rupees you need 1 rupee to pay your cost. So, why not you store 1 rupee as credit with your ah with the element. So, this is very natural and let us do it, but the problem with this approach is that when the table is full only half the later half of the array this part is guaranteed to have rupees 1 stored with them how about the first half ok. So, a natural thing is that you know if this length is i. So, not only store 1 rupee with the currently inserted element also store 1 rupee at the ith index with the element at the ith index from the beginning ok.

So, hence for insertion if it does not involve resize unit. rupees of 3 rupees of money, 1 rupee to pay for its actual cost, 1 rupee to store with the element who is currently inserted and 1 rupee to store with the element at the count minus size by width index from the beginning. So, instead of charging 2 rupees to charge 3 rupees. and this is hallmark of accounting method. You start your analysis with charging small amount and if you if you

are running out of budget then you increase your charge per operation and see whether you can maintain the invariant or not ok.

So, this is the insert process the operation is same, but this analysis for analyzing we are charging every insert procedure 3 rupees 1 rupee is spent for actual resizing and sorry in actual copying if there is no resizing involved and 1 rupee is stored with the currently inserted element and 1 rupee is stored at the or with the item at index current minus size by 2. So, in this way if or there is only sequence of insertion for simplicity let us assume there is no deletion there is only sequence of insertions then when the table becomes full all the elements in the table have 1 rupee with it and hence they can pay for their own movement during resize expensive resize. So, now, let us see the deletion. if the deletion does not involve table resize then

its actual cost is big O of 1 ok. And so, here is how the picture looks like. So, here is again this is size by 2. this is current ok. We use 1 rupee to pay for

the big O of 1 actual cost. Here also you see suppose the table is full and I am keep on deleting and when the size becomes here less than. So, suppose this is size by when the size become less than size by 4, I want all the elements here ok. All the elements here to have a credit with them have rupees 1 stored with them.

So, if the current is more than size by 2, you you can throw away the 1 extra 1 rupee 1 rupee surplus you have you charge 2 rupees you need 1 rupees. So, you if current is more than size by 2, we do not need the 1 rupee surplus. else if here is current if the current is somewhere here in between size by 4 and size by 2 then we should store the extra 1 rupee if this is I this is current. then here at i-th index we should store the extra 1 rupee.

So, that when the table size become size by 4 every element in the table has has 1 rupee with it. So, which which they can use to move themselves else we store rupee 1. with the item at index current minus size by 4. So, this way this was there in the insertion procedure also if the during the insertion procedure if the current is less than size by 2 then there is no element at the index current minus size by 2 because current minus size by 2 is negative then also you have 1 rupee surplus which you can just throw you do not need it. So, this then guarantees the invariant.

So, what is the invariant that at every resize all the elements in the array has 1 rupees. So, let us prove the invariant proof of invariant. So, start from. So, suppose we are doing a resize let us ask when was the last resize when there was a last resize ah then the size of

the table was ah ah the number of elements in the of the table was size by 2 the load factor was half. So, if the current

resize is caused by an insertion process, then there must be at least size by 2 insertions since the last table resize. ok. there could be more because if there is a deletion in between then there may be more insertions, but at least size by 2 insertions and all these insertions are in this region from size by 2 to size and hence by the time the table is full the all the entries in the table has all the elements in the table has 1 rupee stored with them. Similarly, we can show that we can argue that if the current resize is due to the size of

Process then there must be at least size by 4 deletions since the last resize, and every such deletion must have put 1 rupee of credit in the entries with the table currently, ok. So, this proves the invariant, and this shows that both insertion and deletion take $O(1)$ amortized time, ok. So, take it as homework that if you replace the $\frac{1}{4}$ in the deletion process with any constant below $\frac{1}{2}$, then also the insertion and deletion continue to enjoy $O(1)$ amortized time complexity. For example, instead of $\frac{1}{4}$, we can use $\frac{1}{3}$ or $\frac{1}{7}$—whatever any number less than $\frac{1}{2}$, any constant less than $\frac{1}{2}$ in the deletion process results in amortized time complexity of both the operations to be $O(1)$, ok.

So, let us stop here. Thank you.