Second Level Algorithms

Prof. Palash Dey

Department of Computer Science and Engineering

Indian Institute of Technology, Kharagpur

Week – 01

Lecture 05

Welcome to the fifth lecture of second level algorithms course till now we have defined amortized analysis and we have seen three methods for computing the amortized time complexity of various algorithms they are aggregate analysis accounting methods and potential function and we have seen their use in toy examples of in two toy examples One is multi-stat and the other is incrementing binary counter. So, today we will see a full-fledged data structure much more non-trivial than the other two examples and we will study the amortized complexity of the data structure. The name of the data structure is dynamic table. So, let us begin.

So today's topic is dynamic table. So here the operations that we support is insert. So, later we will see the version of dynamic table which supports both insert and delete, but before that let us study dynamic table with only insert key operation. So, how does this table is implemented? dynamic table is implemented as an array.

So, the initial size of the array is 1. we maintain the number of elements currently stored in the dynamic table in a variable called count count is initialized to 0. And let us call the array where we are storing the keys as t.

ok and you also maintain the current size of the array t in another variable called size. So, because we are beginning with an array of size 1, the variable size is initialized to 1. So, what is the insert key procedure?

so if the number of elements in the dynamic table is less than size then I insert the key in the current dynamic table which is the array if count is less than size then t count is key. So we store the key at the location t count. We assume that the arrays start with the index 0. So an array of size say 10 is indexed by 0 to 9.

So, we store the key at t count and then we increase the count or we will increase the count later. If the current if the size equal to count that means the current dynamic table is full then what we do not have the space required to store the key. In this case we create another array of double the current size and copy the old content of the array to the new array and because size has doubled we can now insert the key in the new table. So, I update the size to double, create another array of size of size, this variable size and copy the content

of old array to the first count locations of the new array and then t count is key and we increment the count So as you can see if I store n elements in the array if I perform n insert operations the worst case time complexity of insert is $\Theta(n)$. This is because sometimes we have to perform the expensive copying operation. When the array is full, we have to create an array of double size and copy the content of old array to the new array.

We claim that the amortized time complexity of insert is $O(1)$. So, this means that although there could be some expensive operations sometimes if I perform a sequence of n insert the total time spent on n insert is $O(n)$. proof we have to show that any sequence of n insert operations take $O(n)$ time and for that we will use the aggregate method.

Okay, so what is the total time spent? Total time spent is for every insert, I need bigger of one time to copy the key to the array. So that copying takes $O(1)$ time. So it is $O(n)$ for n many copies. This time does not include the time needed to create a new array and copy the content of old array to the new array.

This $O(n)$ is the total time spent for copying the key being inserted to the location count. the time spent for copying, creating a new array of double size and copying the content of old array to the new array. Let's compute how much time is spent in that part. So observe that because we are only inserting and there is no deletion, so that expensive recreating an array of double size occur only when the array is full, that means only when the array size is some power of 2 we are starting with an array size of 1 then after first operation the array is full and when the second insert happens the array size will be 2 after when that array is full next array size will be 4 and 8 and so on so the total time for copying is then i equal to 0 to $\log_2 n$ ceiling less than equal to instead of equal to let us it is enough we upper bound and when the array size is 2 to the i then the size of the newly created array is 2 times 2^i and the copying takes 2^i time $O(2^i)$ time. So, I have 2^i. So, this is $O(n+\sum_{i=0}^{\log_2 n} 2^i)$. The second sum is at most $2n$. So, this is $O(n)$. So, this proves that the amortized time complexity of insert is $O(n)$. Now if I do not make any change and just allow delete because deletion we can delete simply from a full array no problem while I am deleting an element I do not need to resize the array as in insert.

So, we may think that if we allow delete then deletion takes $O(1)$ time and insertion takes $O(1)$ amortized time. So, the dynamic table with both insertion and deletion takes $O(1)$ amortized time. So but that is wrong. So we will disprove it. Actually I will give it as a homework.

So here is the statement. So, introduce the deletion. Procedure in the data structure which decreases the count variable by 1. So, here is a statement which you disprove.

So, what we can show here is that with this simple deletion, we can actually prove that it is the fact that the amortized time complexity of both insert and delete is O. So, So, we will modify this delete procedure slightly, and then this statement will break, but the statement that the dynamic table data structure with insert key and delete key defined as before has $O(1)$ amortized time complexity for both insert key and delete key. Okay, so this you prove as homework. So, this is homework. The obvious problem with this deletion, this simple deletion, is that if I delete a lot of entries, then the state of the table could be such that most of the allocated space is empty, and we are not using it. So, the problem with this deletion. with this deletion procedure is that it can happen that almost the entire table may be empty.

resulting in inefficient usage of space. So, actually there is a term called load factor which is the ratio of the number of items stored by the size of the table. So, the load factor of the table is the number of items stored in the table by the size of the table. So, with this insertion and deletion procedure the load factor could be all very close to

So, what is the obvious remedy? The obvious remedy is if the load factor becomes too low, we can resize the table and again do that copying thing. And what could be the lower threshold? A natural lower threshold is may be half, if the load factor of the table drops below half, then we make the table size half of the original size. changing the load factor of the new table to be close to 1.

a key if the load factor of the table becomes less than 1 by 2 then we create another table of half the size of the current table and copy all the keys of the old table accept the key to be deleted and decrease the count variable by one so here is the homework disprove that the amortized time complexity

The time complexity of insertion and deletion, as described above, is $O(1)$. This is wrong. So, let us stop here. In the next class, we will see how to change the deletion procedure to achieve $O(1)$ amortized time complexity for insertion and deletion. Thank you.