NPTEL
NPTEL ONLINE CERTIFICATION COURSE

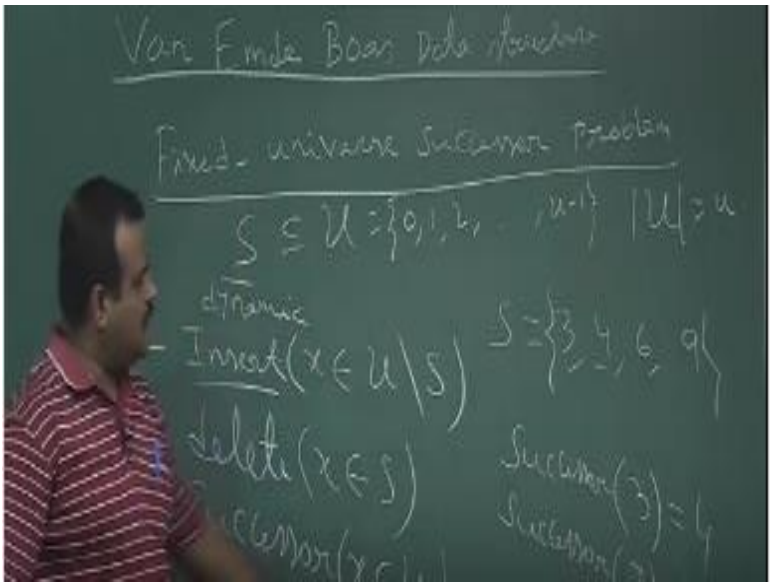
Course Name
Fundamental Algorithms:
Design and Analysis

by
Prof. Sourav Mukhopadhyay
Department of Mathematics
IIT Kharagpur

Lecture 14: Van Emde Boas Data Structure

Okay, so we will talk about really cool data structure which is called Van Emde Boas data structure. So this is the, this is to deal with the problem what is called fixed universe successor problem.

(Refer Slide Time: 00:33)



Fixed universe successor problem, so basically we need to maintain a dynamic set S and this S, the elements are coming from a universe which is fixed, we know the universe. So S is the, S is the subset of the universe, universe are typically u-1, so the size of the universe is basically small u. So we know the elements they are basically from 0-(u-1) and the elements are coming from

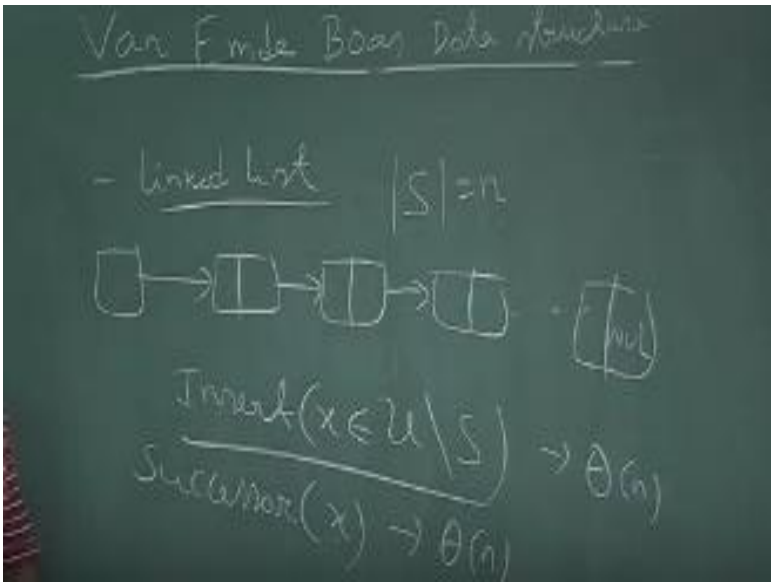
this. So this set is dynamic, so that means we should be able to insert elements so it should be able to perform this operation dynamic operation insert, insert on the element X which is in u, but not in S.

And we should be able to delete an element from S and we should be able to find the successor of an element in u. So that means we want to report the element after X in S. So suppose our S is like this, S is say 3, 4, 6, 9, suppose this is our S at some point of time okay. So now successor say, suppose successor of, we want to find out successor of 3, so successor of 3 is basically the next element in this 4.

Now we can also query this, successor of say 5 or successor of 7, 7 is not in S so it is not necessarily that successor query will be on element which is only in S. It is any element in u, so this is basically 9. So this is the after 7 what is the next element in S, this is the successor. Similarly we can have a predecessor.

So before that what is the previous element in S, so this is the problem so for this problem we need to have a, we want to design a data structure to solve this problem. So can we suggest some data structure for this?

(Refer Slide Time: 03:34)

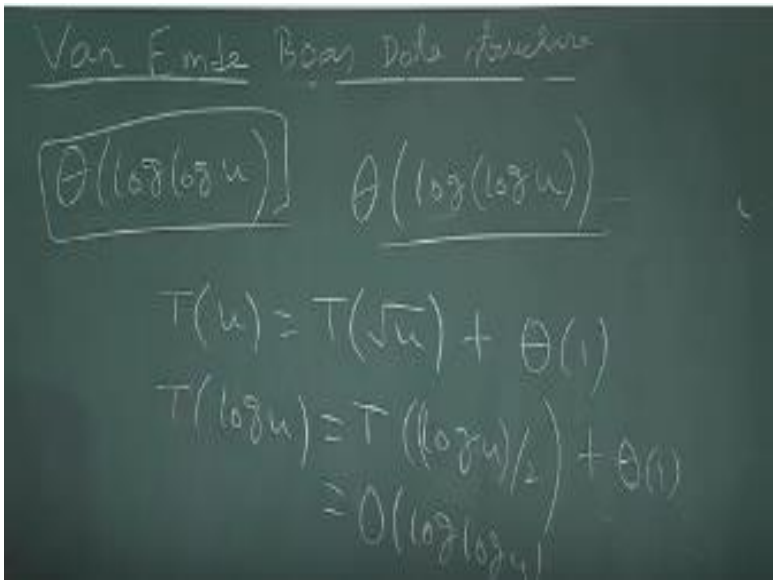


So maybe we can think of linked list, so linked list suppose we maintain a linked list for this case. So once we have element, so we will, we have a top and we just, suppose there are n elements so this is the list and last one is the nil and there are elements and, so now if we use linked list then what is the time for insert, if you want to be insert in the element X which is not in S. So insert means we may have to insert, here all we can have, we may have sorted this.

So insert will take linear time, for delete also if it is not sorted we need to find the element, so that is also linear time. Now the main thing is the successor query. So how much time it will take for successor of, so for successor even if it is sorted we do not know, I mean we may not start from that point.

So that means for successor again we need to scan the list to find out the next element. So successor we take again linear time.

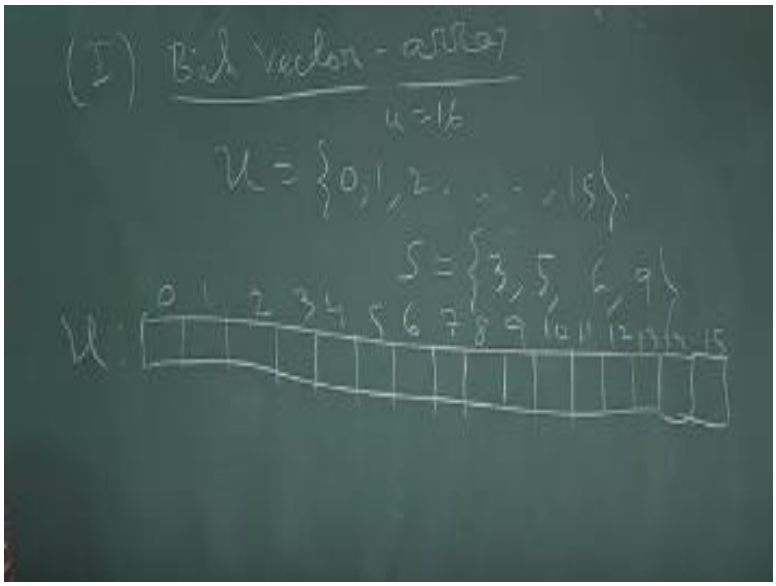
(Refer Slide Time: 05:27)



So we want to something better than this, so can we think about some tree like binary search tree or balance binary search tree, balance BST, okay. So we have a n element and we maintain a BST, I mean balance we can have a red-black tree which is balanced we know.

So we can maintain it, balance binary search tree okay, like this okay. So suppose this is a tree and now suppose we want to find out the successor for this, so successor for this is basically this guys so we have to go to the, from this we move to the root so the time for the successor is $\log^2$ n, okay, so this is if we use the tree specially the balance binary search tree. Now we want to do something better than this, actually we want to do this is in O(log log u) time so that is our target, we want to do the successor query or insert, insertion in the O(log log n).

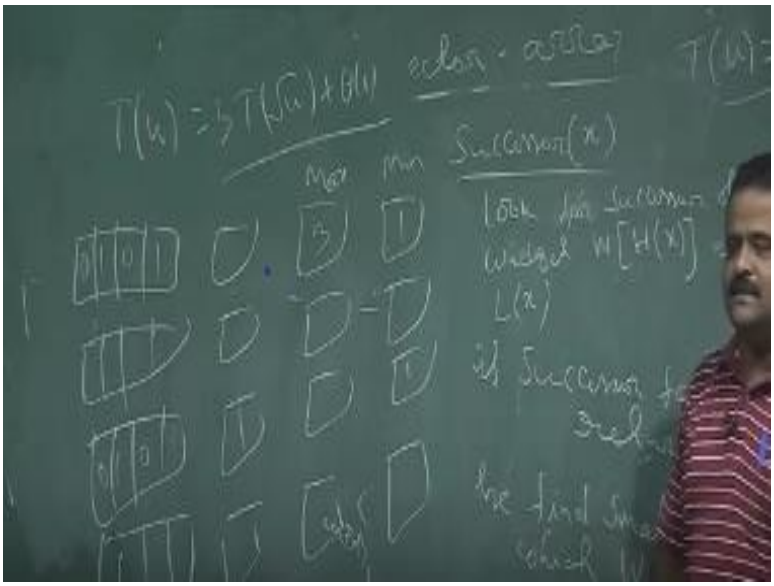
(Refer Slide Time: 07:06)



So how we can get the time? This is log(log u) you can just write this log(log u)) so what form we can get this time log(log u), I mean any situation where our time complicity will be log (log u)) any idea from where we can get this time, so any recurrence thing which can give this solution this? Suppose we have a recurrence like this log n we can get while doing have a tree of size n and we are searching something there I mean binary search, suppose we have log u number of elements and we are doing the binary search then it can give us log (log u). Now we want to do, we want to know what recurrence can give us this solution?

So if we have this recurrence this so if we have this recurrence then the solution will give us log (log u), why because if you just put this in a log u then this will be log u)/2 and then by master on method this is basically log (log u). So we want to get this type of recurrence for this problem so this is the, this is our goal to achieve this recurrence so that we can get the solution as log(log u). So now the question is how we can get that, so you will this is the idea of Van Emde Boas and he has designed this data structure.

(Refer Slide Time: 09:02)



So you start with array is the first step, array or this is called bit vector, this is simpler array, okay suppose our U is say, it's volume is say 16 and our U is say 0,1 to 15 and we have the number s is like this we have taken say 3, 5, 6, 9. Suppose at some point of time this is our number so what we do we just maintain array of size u, so this is basically u array so it is starting from 0, 1,2, 3, 4,5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, okay.

So this is the array, now if a element is present we will put a 1 here otherwise we put a 0, so 3 is present 1,1,1,1 so rest are 0, so this is really a cool data structure, very simple data structure, this is just a called bid vector switch on off, if that room is on light on that means that element is present, okay. So this is the way this is the this is the array we used for this now to insert an element if you want to insert x okay, suppose we want to insert say 8 so what we do we will go to the array 8 and we will put it 1 that is it, so we will go here because 0 so now it is 1 if 8 as to insert so this is the insert operation and delete is also same if we want to delete say 5 so what we do we go to the U[5] and we make it 0 switch on off.

So this we make it 0 like this okay so how much time we are spending here for insert and deletion for insert and deletion which is just a constant time we are just be draw of like this so

now the what the successor what about the successor query suppose want to find the successor for say 0 or1 so successor query for successor query say be we have to scan this array so it will take basically order of two time, okay. But we want to do it in better way so what we do we divide this array into lots this is 16 so we will divide this array into this many blocks and each of size root U the block size is root U, so that will do so how to do that?

So suppose this is our suppose this is our number so 3, 5, 6 and we have 9 okay so we divide so U = 16 so root U is 4 so divide into 4 bit so here, here, here, here so we divide this into 4 bits so this is the first 4 bit this is the second 4 bit these are also called visit or the blocks, okay. This is the first visit we denote at W, 0 this is W1, W2 and then we have W3, so we have basically 4 visit and this is basically W00, W01 like this W03 like this, okay.

And the numbers we have we have 3 over here and we have 5 and 6 00000 and we have 9, 9 over here so sorry 9 is here 000 all are 0, so this is the way we represent as a blocks, so this is the U's port so this how many blocks we have? We have root u blocks and each blocks size is root u, okay. So now how we can insert an element? Suppose we want to insert say 7 so x= 7 so 7 in how we represent 7 in binary?

So 7 is basically 0, 1 as this is basically 4, 1, 1 so 4, 2, 1 so 7 so this we denote high(x) and this we denote by low(x) okay. So high(x) will give as the block number so basically we go to w [H(x)] this will give us the block number and this low (x) will give us the position in that block so this is basically will make it 1, this is the insertion so insertion is basically we will go to that particular block or visit w[H(x)].

So in this case these H(x) is basically 2, so we go to this so high(x) is basic sorry high(x) is basically 0, 1 so it 1 so we go to w1 and then we go to the in that visit we go to the low (x) position, low(x) is basically 3, so we go to the here so this is basically we make it 1 so this is the insertion and similarly to delete will do the same thing so we will go to the this x we make it into 2 part high(x) x root u.

So this is x can be written as this plus low(x) so this is denoting the which block it is belonging and this is the in that block what is the position? So we will go to that particular field so w[H(x)] and then low(x) will make it to 0 for division. So how much time we are spending there? It is just a constant time, now what about the successor, what about the successor query? Okay, now suppose we have this situation suppose we have here it is 0 and suppose we have here it is 1.

So S is now so 3 then we have 5, 6, 7 and then this is 15 suppose S is like this okay, now suppose we want to find out the successor of say 7, x is 7 okay so what we will do? So we first make it into two part high(x) and low(x) so we go to that W[H(x)] that particular digit visit and then we find out the successor in that visit if there is if this is last element so we know successor exit here.

So what we do? Then we go for the next block where we have, we can find out the region, but this block is empty, so, so but we have to check it there is no way we can because we do not know whether this is empty or non empty, then we cannot find here then we go for then this is the, so how we can augment the data structure so that will not go into this block..

So for that what we do, we will just simple augmentation in the data structure we will put extra bit of information which is non empty, so this is the non empty bit, so what we do if there is a 1 we put here 1, 1 otherwise if there is no 1 we will put 0 and we will put 1, so the I will go is now, so what we are doing suppose we are finding the successor(x) so what we do, we look into the, look into loop for successor of x within, within the width it or within width it W[H(x)]. We will look within the width it W[H(x)] and starting from, starting from L(x) so this is the first step, okay.

Suppose you want to find the successor of say 1, successor of 1 or 2 so what is the H(x), H(x) is 0 and L(x) is basically 0, 1, so lower so this is basically 0,0,1,0. So L(x) is 2, so we will start from this width and we will start from this L(x) after that we will find the next one, we will find the successor in this block. Suppose we put this is 0 then we have to look at the next non empty block if successor found then we successor found then we return it, okay.

Otherwise we will just look at the else, look at the smallest, smallest i greater than i(x) for which W[i] this width is non empty, okay. so that means we will, if we are not finding here we will look at the next visit where we have a non empty so this is non empty, so we will look at here, so that is the idea, so this is 2 and then in 3 what we do, then we find the smallest element in W[i] so suppose we know this is the non empty so we will written as smallest element so this is has 1 so it will return this, so this is the code.

So this is basically what is the time complexity for this, so we are looking for a successor over here, so this is size of this is √u, so this is of Θ√u and we are again we are looking for a non empty, non empty visit so that means this is also successor problem, so we will look after that we are looking for what is the next one. So that is also, so this is also order of √u and then again in that block we are looking for a successor starting from so this is the successor starting from minus infinity to this, okay so this time is basically order of √u.

But we won this time is log, log u so for that we need to achieve this reoccurrence u= T√u+ Θ1 so how we can achieve the reoccurrence so we need to have a recursive call for this alga so can you define a recursive call for this so what is the recursive note for this alga so can you have a recursive call for this yeah so this is basically successor.

Or successor of say in the w of h of ax in this array after low of x so this is a successor call and here also this is also successor call where in the that non mt array non mt array so this is basically successor call of yeah so let me right again successor on the non mt non mt array after this h(x) so this is also successor call and the last ones once we know this is non mt so for this, this is non mt, now here what we looking for we are looking for the.

First non 0 element so that is basically also successor call successor are of sort of minus infinity in that visit w(i) okay so basically this is a three successor call so the recurrence will be T(u) = 3 T(√u + Θ(1) but this will not give us the log, log u so we need to reduce the successor call 2 log there 3 successor call so how we can reduce the successor call 21 call the 1 successor call now see if we cannot find here then if we can find the successor here then we are not.

Going to this to 11 okay so and if we are not finding here then we are going for this two call so how we can reduce this to one call so idea is you want to do the more augmentation on the data structure so along with the non mt beat we want to store the maximum and the minimum maximum position up to higher this one exist, so if this is the array so we have this and this is non mt bit along with this we have a maximum max bit max bit means if this is a 101 so that mean this max bit will be 3 over here.

Okay, so this is the maximum position where one is so what is the benefit of this so for this if we know the maximum and if we all low of x is better than maximum then we know that there is nobody here which is a successor so we will not call this successor call for the first case then we will go for successor call over here in this non mt bit okay now if we know the maximum is this maximum bit is better than low of x then we will do the only one successor.

Call there because we know there is a successor okay but if there is no successor they are then we have to look at the first non mt bit suppose this is one and then suppose 1 0 1 now here we need to get the successor so how we can get that so we will just do further augmentation we will put minimum also because we already made a successor call so we will put the minimum so minimum means this minimum is one this minimum is also one.

Then we the so this, this will give us the successor of x so we will just look at this minimum and we will return that so basically we are doing the all the one successor call if low of x is less than maximum the we are looking for the successor in that log that is one call if not then we are taking it fall to get the successor in this, I mean non mt element in this then we going to the particular visit and then we are returning the particular minimum element over here so only one successor call we are doing in any case so that is the reason or.

Reoccurrence is last one so this will give us the solution log, log n so this is the idea of hand mt boss so this is really, really cool data structure and we are by this augmentation we are getting this time complexity okay thank you.