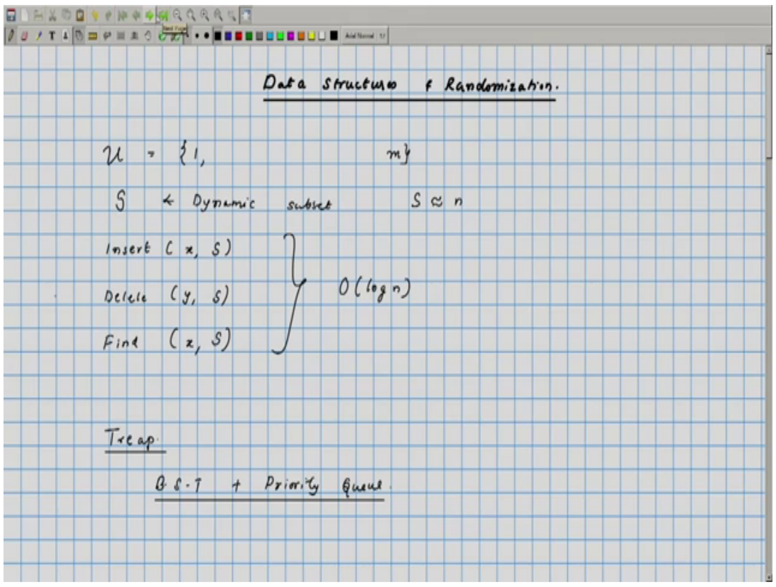
Randomized Algorithms

Prof. Benny George Kenkireth
Department of Computer Science & Engineering
Indian Institute of Technology, Guwahati

Lecture - 34
Treaps

In this lecture, we will see how randomization helps in design of good data structures ok.

(Refer Slide Time: 00:34)



Data Structures + Randomization.

$\mathcal{U} = \{1, \quad m\}$

S ← Dynamic subset $\quad S \approx n$

$$\left.\begin{array}{l} \text{Insert } (x, S) \\ \text{Delete } (y, S) \\ \text{Find } (x, S) \end{array}\right\} O(\log n)$$

Treap.

B.S.T + Priority Queue.

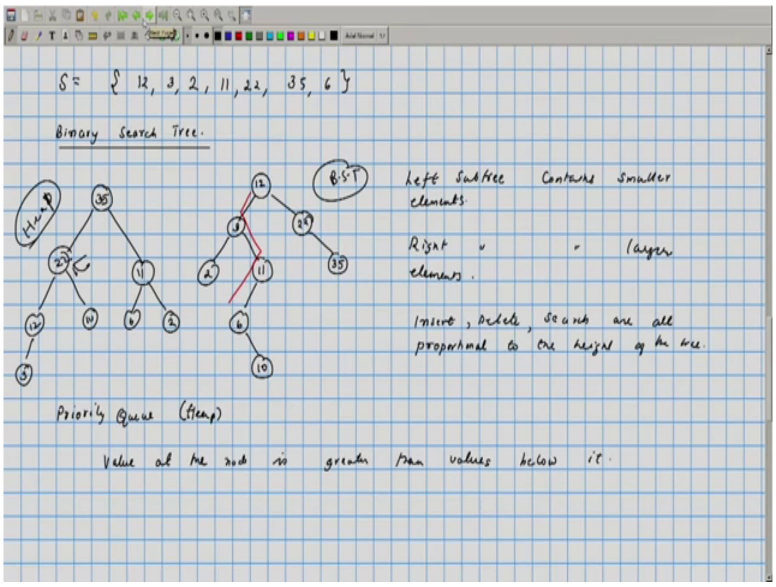
So, let us look at the problem that we are interested in solving today. So, there is a universe u consisting of elements let us say integers one to some m and then there is a dynamic subset of u and let us say that the sizes that is roughly n ok. So, it is varying so it does not a fixed size, but at any given time let us say it is very close to n.

Now we want to do certain operations on this data the operations that we will be doing are insert an element let us say x into the set S to produce a new set and then delete particular element y from the set, then we will also have search or find ok. So, if you are given a key you want to know whether that element is present in S or not.

So, these are the operations that we want to implement and we want to do this efficiently and by which we mean we want to do all these operations in order login and we want, so this is our objective and let us see how data structures and randomization can help us do this. So, what we are what we will do here is we will look at a new data structure called as treap ok. So, treap is nothing, but a marriage of binary search tree and priority queue. So, let us understand what these components are.

So, treap is nothing, but binary search tree plus priority queue or a heap ok. So, let us understand what are these individual operations, what are these individual data structures and how we will do the insert delete and find operations on them ok.

(Refer Slide Time: 02:55)



S = { 12, 3, 2, 11, 22, 35, 6 }

Binary Search Tree.

Left Sub tree Contains Smaller elements.

Right " " larger elements.

Insert, Delete, Search are all proportional to the height of the tree.

Priority Queue (Heap)

Value at the node is greater than values below it.

Let us take a sample data ok. So, suppose this is our input data, so, this is our S and a binary search tree will be something like this ok. So, if this is our input, then the binary search tree corresponding to that would be something like this ok.

So, the key property of binary search tree is that, left sub tree contains smaller relevance, then whatever is at the node and right sub tree contains larger elements ok. So, suppose you have a partial sub tree like this and if we wanted to insert the element 10 into it ok. So, you check at the root, so 10 will essentially be on the left side of 12 ok, so you will go here and it is greater than 13 so it will go here and it is smaller than 11, so we will go again to the left and it is larger than that. So, basically you will create a new node wherein 10 goes

So, the insertion is easy and if you wanted to delete you can essentially unroll the process of insertion. So, suppose you wanted to delete 11, it would mean that you can just connect 6 in the place of 11 ok, if you had two nodes as children you will have to do a slightly more complicated operation, but those can be done. So, binary search tree is essentially a data structure where every node on the left sub tree contains smaller elements and everything on the right contains larger elements than whatever is present at any particular node, those property should be true of every individual node that is called as a binary search tree and you can use this fact that insert delete and search are all proportional the time taken by these operations are proportional to the height of the tree.

The second data structure that we would see is something called as a priority queue or a heap ok. So, in heap the key property is the heap property which says value at the node is greater than values below it ok. So, if you look at the binary search tree there are children which are grade which have larger value than the values of node, but priority queue or heap that property that is not true ok.

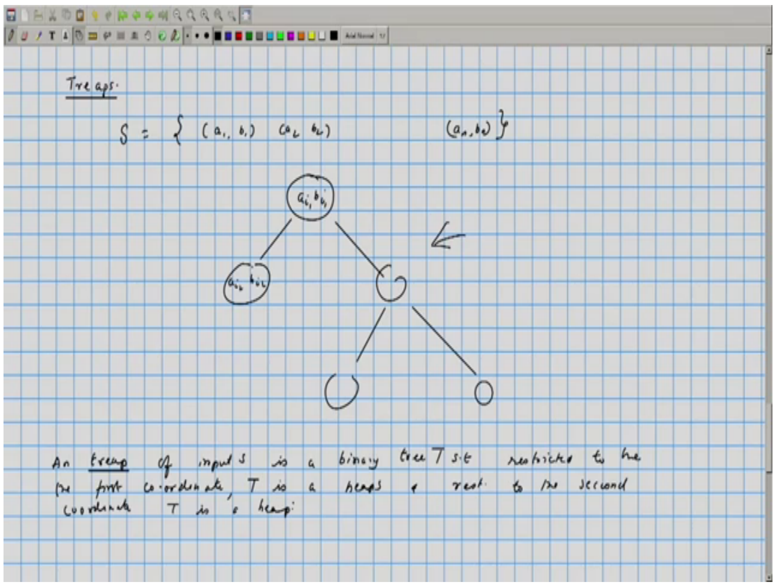
So, corresponding to this input the one possible heap is as follows the largest would of course, be 35 and let say that the moment we would not bother about how these were constructed ok. So, if you look at this diagram on the left, that is a heap and this is a binary search tree ok. So, if you look at any node for example, if you look at 22 you can note that it is children are of value smaller than whatever is there in the node, this is true not just for the node 22, it is true for every node in this particular tree and such trees are called as heap ok.

Given any input data you can convert it into a heap in linear time and the heap need not be unique I mean it is usually not unique there could be given the same data you could make multiple heaps with the with that given data the same is true for binary search tree, but if you are inserting in a particular way then of course, there will be a unique binary search tree ok.

But for if we look at the data as just a collection or a set of numbers there are multiple binary search trees and multiple heaps. So, now, when we are given a collection of data we going to convert them, we are not going to store them a size or binary search tree or as heap, but instead we will combine these two and the combination essentially ensures that the operations that we wanted to do, namely the insert, delete and find these can all be done in expected o log n and the expectation is over some particular choices.

Those choices are not really the input, but it is based on the coins random coins that is flipped while the algorithm was run ok. So, let us now that we understand what are binary search tree and priority queues or heaps we can define what are called as treaps ok.

(Refer Slide Time: 09:24)



An ***treap*** of input S is a binary tree T s.t restricted to the the first co-ordinate, T is a heaps & rest. to the second coordinate T is a heap.

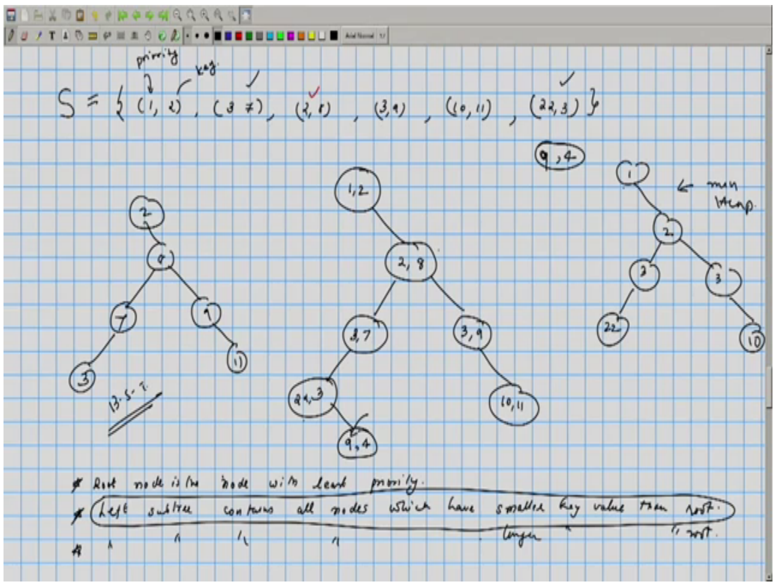
So, the input data is slightly different from the data for constructing a heap while constructing a heap or binary search tree, the input was a set of numbers here while constructing treaps the input is basically a set of pairs of numbers ok.

So, let us say S is going to consist of numbers of the consists of pairs, so a 1, b 1 a 2, b 2 and a n b n ok. So, on this particular set we want to construct a treap, so treap was nothing, but a tree where the individual nodes are some ai bi ok, so let us say a i 1, bi 1 ai 2, bi 2 and so on ok.

So, the input data is kept in some particular way and there are going to be this tree is going to consist of precisely n nodes, the input consists of n pairs and we say that it is a treap if when restricted on the first coordinate, if you just look at the ais look at the diagram look at the tree and erase all the bis you will get a tree with just ais just the first coordinates.

Restricted to first coordinate, the tree should be a heap and restricted to the second coordinate this should be a binary search tree ok, so that is a treap. A treap on input s is a binary tree, such that restricted to the first coordinate the binary tree T is a heap and restricted to the second coordinate T is a heap. A priory it is not even clear that given data we can organize in this way, but it is easy to see that given a neat set of pair of numbers there is a unique treap corresponding to it.

(Refer Slide Time: 11:59)



* Root node is the node with least priority.
* Left subtree contains all nodes which have smaller key value than root.
* " " " " " larger " " root.

So, let us take an example given assume that the numbers the pairs are all unique ok. The first coordinate we are going to refer it to as the priority because priority is what is required in a heap and the second coordinate is what we will call as the key ok. Now, we know that the topmost element is the element with the minimum priority, so this should be 1, 2 ok.

So, we are going to look at a min heap, so 1, 2 should be this element and the elements on the left should be on the left child should be everything which is smaller than 2, when you look at the keys and there is no such element, so there is nothing on the left. So, everything else is on the everything is going to be on the right and if you look at the remaining elements the least priority is 2 comma 8, so that is going to be here. And on the left side will be all elements whose key value is less than 8, that is basically 3 comma 7 and 22 comma 3 and amongst them the root is going to be the one with the smallest priority, so that is going to be 3, 7 and as the other node is 22 comma 3 and that will be the left child because 3 is smaller than 7.

So, 22 comma 3 will be here and all these other three elements will be on the right side of 2 comma 8. So, 2 comma 8 is already taken care of, so there is 3 comma 9 and 10 comma 3 and out of them the least priority is 3 comma 9. So, this node is going to be 3 comma 9 and the only remaining node 10 comma 11 is going to be the child of 3 comma nine and since 11 is greater than 9 this is going to be the right child, so 10 comma 11.

So this is a treap I saw if you just look at the first coordinates it is going to of this kind, note that any node is smaller than it is children than any of it is children therefore, this is a min heap. Whereas, if you look at the second coordinate 2 followed by 8 followed by 7 followed by 3 and comma 11. If you look at any node everything on it is right is larger than that and everything on the left is smaller ok, so this is a binary search tree ok.

So, given any data there is precisely 1 treap corresponding to it, that is because if you assume that the keys and the priorities are distinct the root node is going to be the node with least priority. And then the left child or the left sub tree contains all nodes or all pairs which have smaller key value than root and the right sub tree contains all nodes which have larger key values than the root ok. And this sub tree there can be precisely one way that can be organized as a treap and same for the right sub tree. So, once you make this sub trees or sub treaps there is one way you can combine it and that is going to give you a unique tree.

So, given any input data that is precisely I mean if you assume that the priorities and the keys are all distinct where the priority could be a completely different set from the keys, but as long as those sets are themselves unique when there is no repetitions amongst the keys or amongst the priorities there is only 1 treap corresponding to it.

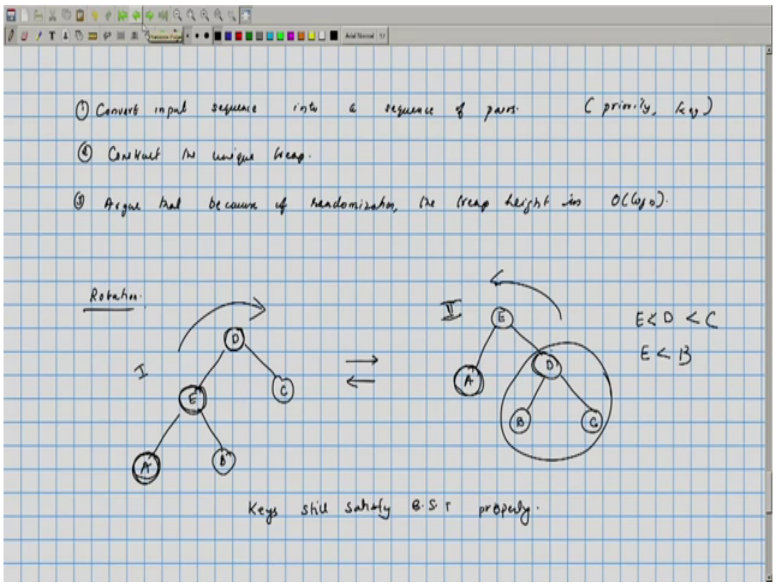
Now, we could do all the insert operations or delete operations or the find operations based on the values of the key ok. So, here what we really want the problem that we really want to solved is, we have some particular in collection of numbers what we will do is we will convert each of those numbers into a pair ok, the second coordinate will be the number and the first coordinate will be the priority and this priority is where we will introduce the randomness.

So, what we will do is given any collection of numbers we will choose a priority a random number between 0 and 1 chosen uniformly at random and that will be the priority of this particular number ok. So, you can say that since we are choosing and uniformly at random between 0 and 1 with probability one every number will get a unique priority ok. So, if the keys where themselves distinct the random choice also is not going to introduce any identical priority elements ok. So, given a collection of input we going to choose a random number between 0 and 1 as the priority for that and then generate the heap corresponding to it ok.

Once we generate the sorry not the heap, but the treap corresponding to it once you have the treap corresponding to it we want to see if the insert operations, delete operations can be done quickly it will be done in order log n and whether the search operations can be done in log n. We will see that all these operations the time taken to do this depends on the height of the treap.

The treap is the tall treap then it is going to take a long time, if it is short it is going to be the operations are going to be fast, but we will arguers and because of our randomization the treaps height is going to be bounded by log n in an expected sense ok.

(Refer Slide Time: 19:58)



1. Convert input sequence into a sequence of pairs. (priority, key)
2. Construct the unique treap.
3. Argue that because of randomization, the treap height is $O(\log n)$.

Rotation.

Keys still satisfy B.S.T property.

So, in a nutshell what we will do is convert input sequence which was a sequence of integers into a sequence of pairs ok. So, if the input if you think of them as keys, we will convert it into priority key pair and then we will construct the unique treap if you are introducing a new element that will also result in a unique treap which can be incrementally computed from the existing treap and then we will argue, that because of randomization the treap height is bounded by o log n ok.

Now, when we are inserting and deleting we have not yet argued that the insertions can be done in time proportional to the height of the tree. So, let us first look that for that there is an important notion called as rotation. So, let us say that this is a particular treap and where there is a value and there is a key and a priority this is the priority and this is the key ok. So, let us say that we have some tree a treap at some point of time. So, this is a root node and it has children ok.

So, let us say that one of the child was a child was A the other was A and the other was C. So, these are 2 nodes we can convert this via rotations we will call that as a left rotation and the right rotation ok. So, if you do the left rotation we will get another tree, so let us call this as D and E ok, note that D and E are basically nodes and ABC are all there are themselves trees ok, if you right rotate what happens is I mean giving a rotation to the right side what happens is E comes on top, D goes down, A comes up and D has 2 children now C and B ok.

And if you left rotate this D comes up, E goes down and A goes further down, B attaches itself to E ok. So, this is an operation which is this left rotation and right rotation are well defined for binary search trees, in a priority queue also we can think about these operations, but they might destroy the priorities. We will see whether treaps we can do these kind of operations on treaps ok.

So, let us say that if you rotate in this kind the there were some existing values at these particular nodes keys ok. So, key observation is that keys still satisfy BST property or the Binary Search Tree property because everything in A should have been lesser than everything in E which is clearly satisfied in both trees and tree mean if you look at this as called this tree 1 and tree 2. In tree 1 if you look at node A; node A was less than E and that is still valid and every value in D mean, so we look we can look at it node by node we will just do a couple of examples.

If you look at node E A consists of smaller elements and B consists of larger elements that property is still true because A is in the left sub tree, B is in the right sub tree. Now note that D is going to be greater than E and C is going to be greater than D ok. So, D is greater than E and C is greater than D ok, so both D and C are greater than E and therefore, in the tree 2 you have both D and D and B is anyway greater, so E is smaller than B.

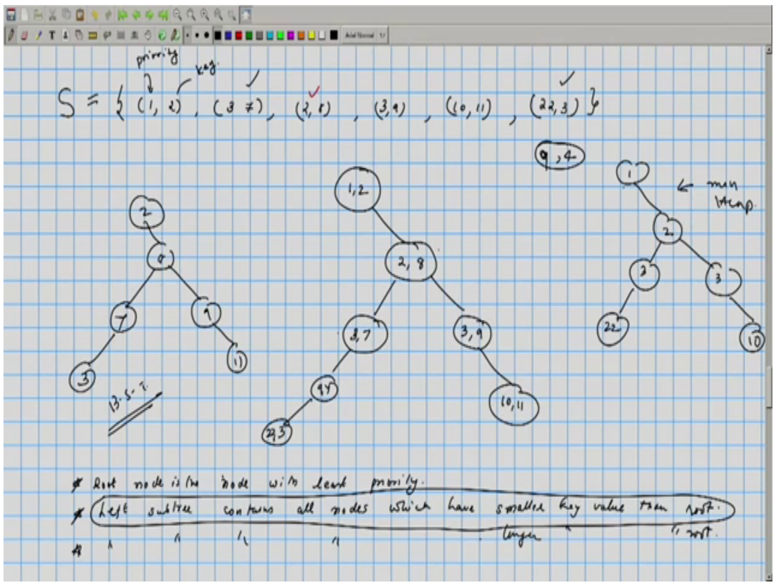
So, all those relationships are satisfied for E, so you can do this for all possible combinations and you can verify that these rotation operations, if you rotate to the right it will basically preserve the BST properties and if you rotate to the left also the BST properties are maintained ok.

Now, when we are inserting and deleting we can use these operations to ensure that at no point we require more than the height the I mean I mean we can maintain the tree property by insuring by doing these things properly let us let us see that. When you if you had to just maintain the binary search tree property we could just insert and when you insert the binary mean you will insert it ensuring the binary search property, but the element that is inserted may destroy the treap property. For example, in this particular data if you think of S as the input data and if you wanted to insert an element, so if you wanted to insert an element 4 comma 9 ok.

So, here is an element whose key value is 4 and priority is 9 ok. So, try to insert based on 4 what happens is we just looking at the key value it is greater than 2 so it should be on the left sub tree, on the right sub tree it is less than 8, so it should be on the left it is less than 7 it should be on the left and 4 is greater than 3, so it should be on the, so this is going to be 4 this it is priority was 9 ok. For this node, the treap property is not valid because here is a node whose priority is 9 and that is below a node whose priority is 22, so the priority has to be adjusted ok.

So, what we can do is we can rotate accordingly, we can do these operations rotations left rotations and right rotations ok, these operations do not kill the BST property, but we can use this to adjust the priorities let us see how that is done ok.

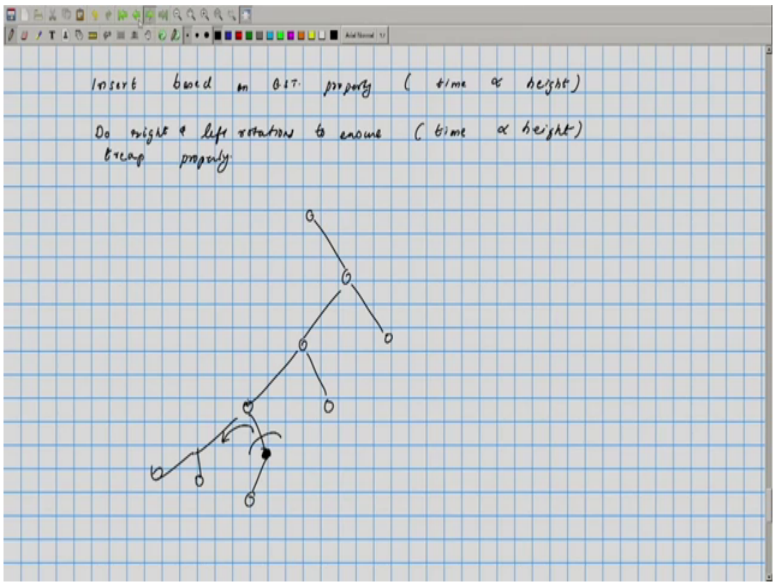
(Refer Slide Time: 27:59)


priority
key
S = { (1, 2), (3, 7), (2, 8), (3, 9), (10, 11), (22, 3) }
9, 4
min Heap
BST
* Root node is the node with least priority.
* Left subtree contains all nodes which have smaller key value than root.
* larger root

So, from here we could move you could do a left rotate and therefore, 9, 4 goes up, so 9, 4 goes up and 22 comma 3 comes down ok. Now this ensures that the BST property is maintained at this particular node between these pair of nodes the BST property and the tree property is valid, but now if you look at let us say 3, 7, 9, 4 ok. So, if here every I mean every node it is BST property as well as the tree properties holding because 2 and greater priority and greater priority and greater priority.

So, maybe if this node had a higher value you might have to do a right rotate. So, basically by a sequence of left and right rotations you can ensure that the treap property is maintained ok, so in a nutshell what we have argued is the following

(Refer Slide Time: 28:57)



Insert based on BST property (time $\propto$ height)

Do right & left rotation to ensure (time $\propto$ height) treap property.

Insert based on BST property, when you do this the time taken will be proportional to the height and once you have inserted a particular pair the tree property need not be maintained. So, we will do appropriate right and left rotations to ensure treap property.

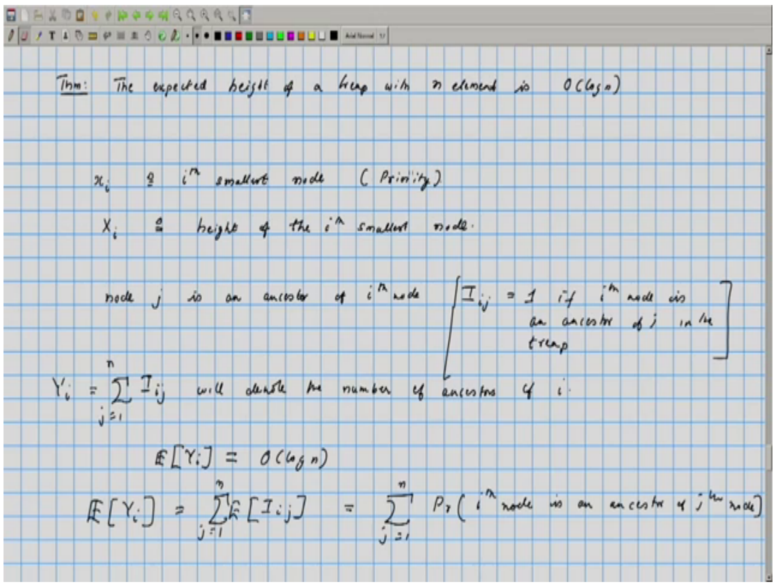
Treap property in the sense the BST property comes for free because we had inserted based on the BST property. The rotations either in the left side or the right side do not affect the BST property, but we are using those to adjust the priorities. So, one by doing a sequence of left and right rotations we can ensure that the tree properties also maintained ok. So, suppose you had some particular node like this ok.

So, let us say this was your temporary treap and then we wanted to insert an element the BST property ensures that the element was added here ok, maybe this node it had to go somewhere else, so that the treap property is maintained. So, you look at these two nodes and decide whether it is to be rotated right or not and then if it is rotated right what happens, this node becomes the newly added node goes up and then maybe depending upon these nodes you might have to rotate it left or right and maybe this goes up.

So, the total number of operations that happen this is going to be bounded by the current height of the tree ok. So, this operation also in this is the time taken is proportional to the height ok. So, what this means is any insertion in the treap can be done in time proportional to the height of the tree and deletion is exactly same as insertions just delete an element and if it is an intermediate element you might have to, so that the treap property is maintained.

So, all those operations this is similar to the usual binary search tree operations with these rotations and they take time proportional to the height of your tree. What we need to argue is that, the height of the treap in the expected sense remains small in the sense it is going to be equal to o log n, so that is what we will show next.

(Refer Slide Time: 32:05)



Thm: The expected height of a treap with n element is $O(\log n)$

$x_i \triangleq i^{th}$ smallest node (Priority)

$X_i \triangleq$ height of the i^{th} smallest node.

node j is an ancestor of i^{th} node $\left[I_{ij} = 1 \right.$ if i^{th} node is an ancestor of j in the treap $\left. \right]$

$Y_i = \sum_{j=1}^{n} I_{ij}$ will denote the number of ancestors of i.

$\mathbb{E}[Y_i] = O(\log n)$

$\mathbb{E}[Y_i] = \sum_{j=1}^{n} \mathbb{E}[I_{ij}] = \sum_{j=1}^{n} P_r(i^{th}$ node is an ancestor of j^{th} node$)$

So, this is the theorem the expected height of treap with n elements. So, let us say we have done a sequence of insertions and deletions and finally, we have n elements and the expected height of the treap with n elements is o log and therefore, all these operations insert delete, so it is etcetera can be done in o log n expected time ok.

So, how do we prove such a statement? This is happening because of the random choices that we have made, the priority queue that was there with respect to the first coordinate. That priority queue it is height is never going to be more than o log n and in an expected sense that is what helps us to maintain the overall height.

So, let us introduce some notations, so x i will denote the ith smallest node in terms of when you say ith smallest this is in terms of the priority. So, this ith smallest node in terms of the priority not on in terms the key value will be somewhere in the tree we will look at it is expected height ok, so let us denote it by random variable.

So, let us say capital X i we will define as the height of the ith smallest node, this is a random quantity because it depends on when this element was added and so on ok. We are assuming that each element gets a random priority and that priority is being used to create a point pair the first coordinate being the priority and second point being the key and X i is the height.

Let us introduce some more random variables. So, we are looking at the event that node j is an ancestor of ith node ok. So, let us indicate this by the random variable I ij. So, I ij this is equal to 1 if ith node is an ancestor of j in the treap and summation j going from 1 to n I ij will denote, the number of ancestors of i we want to show that, so if you denote this by the random variable let us say y i, we want to show that expected value of y i is equal to o log n.

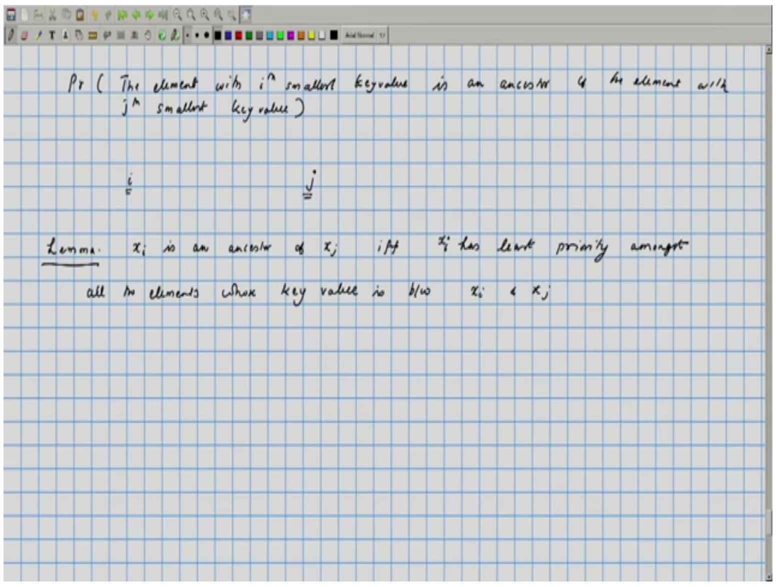
The expected value of y i it is approximately log n, it is bounded by log n that is what we want to argue. This is some of certain random variables so, expectation of y i by linearity of expectation is the expectation of I ij summation j going from 1 to n ok. So, now what we want to know is the expectations of this individual I ij, but the expectation of an individual indicator random variable is nothing, but the probability of the event that indicates.

So, this is summation j going from 1 to n probability that ith node is an ancestor of jth node, ith node or x i is an ancestor of the jth node ok. So, ith node here means that particular pair which had ith smallest priority and jth node would means jth smallest priority ok.

So, we have to slightly modify our argument here we were looking at the smallest in terms of priority, but let us look at I mean we could do this entire analysis on the basis of the key values as well ok, ith smallest node means the key value is ith smallest and jth smallest means the key value is the jth smallest ok.

Now, so, these were our numbers the key is where let us say from some particular set the ith smallest key value that node is what we call as x i and the jth smallest key value that is what we call as I mean x j and so on.

(Refer Slide Time: 37:59)



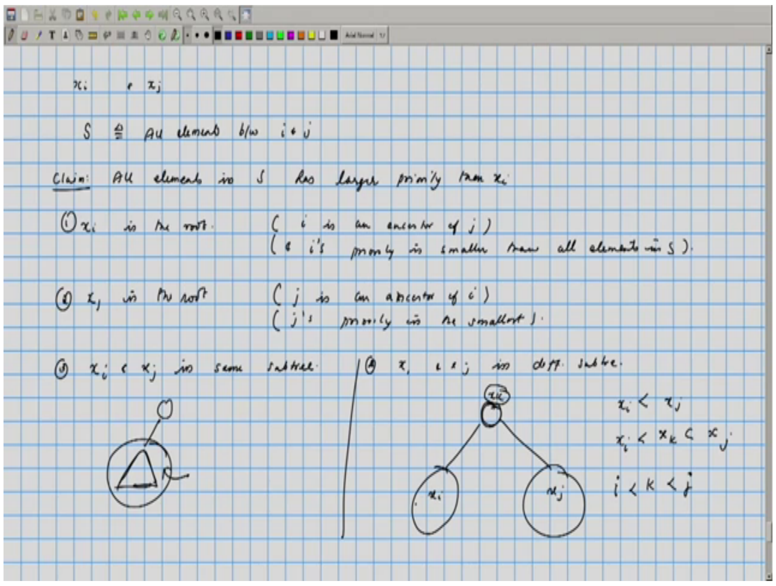
Pr (The element with i^{th} smallest keyvalue is an ancestor of the element with j^{th} smallest key value)

$\underset{=}{i}$ $\underset{=}{j}$

Lemma: x_i is an ancestor of x_j iff x_i has least priority amongst all the elements whose key value is b/w x_i & x_j

So, probability that the ith smallest key value is an ancestor of the element with jth smallest key value this is the probability that we need to ascertain ok. So, let us look at these elements i and j ok, so small and we have lot of numbers in between as well ok. So, we will have an lemma which will help us estimate this probability ok, x i is an ancestor of x j if and only if i has or x i has least priority among all the elements whose key value is between x i and x j.

(Refer Slide Time: 39:28)



x_i & x_j

$S \triangleq$ All element b/w i & j

Claim: All elements in S has larger priority than x_i

① x_i is the root. (i is an ancestor of j)
(& i's priority is smaller than all elements in S).

② x_j is the root (j is an ancestor of i)
(j's priority is the smallest).

③ x_i & x_j in same subtree. | ④ x_i & x_j in diff. subtree.

$x_i < x_j$

$x_i < x_k < x_j$

$i < k < j$

So, we have these nodes x i and xj and all elements appearing between i and j let us call that as a set S ok. So, S is a set of all elements the key value is between i and j what we want to prove is all elements in S has larger priority than x i. So, we will split the proof into 4 parts we added treap and suppose x i was the root by virtue of it being the root, its priority is going to be surely smaller than the priority of all the other elements ok, in particular all the elements in S will have strictly smaller priority ok.

So, if x i was a root then we know that, i is an ancestor of j and i's priority is smaller than all elements in S ok, so that is case one. Now if j is the root it is the same argument x j is the root, then we know that j is an ancestor of i and therefore, by virtue of j being the root it is priority is the at least and therefore, it is priority is smaller than all elements in S is the smallest.

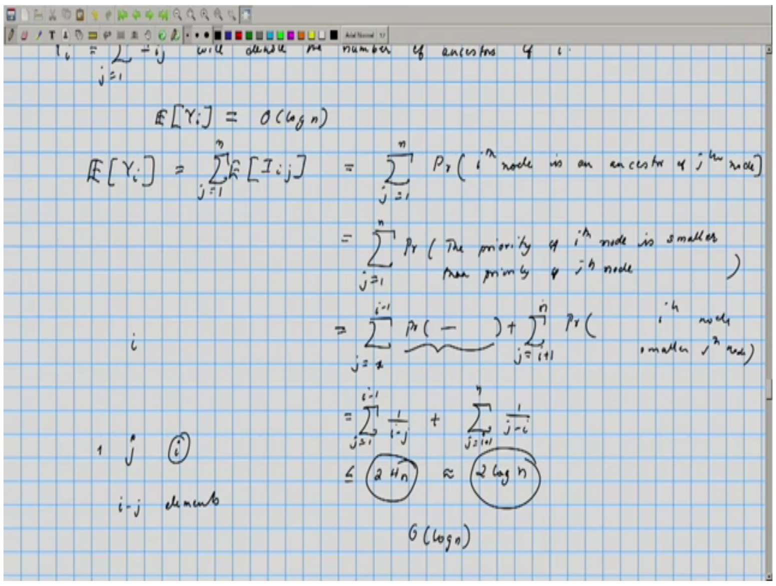
Now, if it is not root there are two cases they could be in the same sub tree or they could be in different sub trees ok. So, the third case x i and x j in same sub tree so; that means, there is some root and then x i and x j is in this sub tree ok, by recursion you can say that the same argument should work ok. Because now you have a smaller set of numbers these numbers they are strictly smaller than that and x i and x j are elements here and if there is an ancestor descendant relationship between x i and x j we can restrict our argument to the smaller sub tree. If they were in different sub trees what do we do? So that is our 4th case ok.

So, let us say the root node is some x k and this is a sub tree containing x i this is a sub tree containing x j ok. Now one of them is on left and the other is on right, the elements on the left by nature of the BST we know that I mean if x i is on the left, xi is less than x j and in particular your x k is going to be lying in between x i and x j. If you look at the key values the key value of x i is going to be less than the key value of x k and the key value of x j is going to be greater ok. So, this would mean that I is less than k is less than j.

Now, k is an element in between i and j and that has the least priority from the set S ok, x i is the node which is having least priority and therefore, if you look at the elements in between x i and x j, x i is not an ancestor of x j; xj is not an ancestor of x i and we have neither of them neither x i nor x j has the smallest priority it is some other element and therefore, that takes care of all the 4 cases ok.

So, this means that whenever some element is an ancestor of some other element that particular node has least priority. So, now, all that we need to estimate is the probability that a particular node has least priority amongst a collection ok. So, that would help us calculate the expectation ok.

(Refer Slide Time: 44:42)



$Y_i = \sum_{j=1} I_{ij}$ will denote the number of ancestors of i.

$$\mathbb{E}[Y_i] = O(\log n)$$

$$\mathbb{E}[Y_i] = \sum_{j=1}^{n} \mathbb{E}[I_{ij}] = \sum_{j=1}^{n} Pr(i^{th} \text{ node is an ancestor of } j^{th} \text{ node})$$

$$= \sum_{j=1}^{n} Pr(\text{The priority of } i^{th} \text{ node is smaller than priority of } j^{th} \text{ node})$$

$$= \sum_{j=1}^{i-1} \underbrace{Pr(-\quad)} + \sum_{j=i+1}^{n} Pr(i^{th} \text{ node smaller } j^{th} \text{ node})$$

$$= \sum_{j=1}^{i-1} \frac{1}{i-j} + \sum_{j=i+1}^{n} \frac{1}{j-i}$$

$$\leq \boxed{2H_n} \approx \boxed{2\log n}$$

$$\Theta(\log n)$$

i

$1 \quad j \quad \boxed{i}$

$i-j$ elements

We needed to compute this probability; probability that the ith node is an ancestor of the jth node. So, this is equal to summation j going from 1 to n, probability that the priority of ith node is smaller than priority of jth node ok.

So, if you check the case that i is less than j, so this summation we can split it into 3 parts summation j going from 1 to i minus 1 probability that the priority of the ith node is less than priority of jth node plus summation j equals i plus 1 to n, probability that the priority of the ith node is smaller than jth node and this is what we because if you take the case where j equals I that probability is 0.

And this probability ok, so here in this side we have some particular j and amongst the set 1, so let us say this is one I and here is your j. So, you have i minus j elements ok. So, i minus j elements which are competing to have the least priority, the priorities where chosen randomly. So, every element has equal chance of being the lowest priority element.

So, i being the low least priority happens with probability 1 by i minus j. So, this summation plus summation j going from 1 to n 1 by j minus i when j is greater than i this is going to be 1 by j minus i. So, these probabilities have to be summed this is going from j going from 1 to i minus 1 ok.

So, each of these terms you can upper bound by the harmonic number, so this is going to be less than 2 times H n and H n is approximately log n, so this is 2 log n ok. So, what this means is that the expected height of any particular node is bounded by o log n and that completes the proof we will start the new topic on hashing in the next lecture.