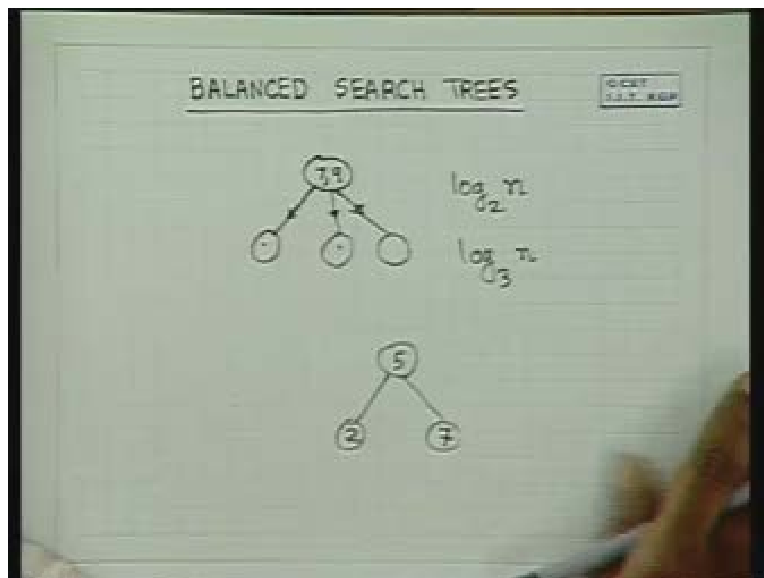
Programming and Data Structure
Dr.P.P.Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 25
2-3 Trees

The topic of this lecture is balanced search trees. And as we have been discussing that in order to make search trees effective, we must try and get the worst case height to be log n where n is the number of key values and nodes which are stored in the search tree. And we mentioned among binary search trees, the AVL tree or the red black tree or the types of data structures where the basic operation after every insertion or deletion is reorganization or process of balancing.

Now search trees can be other types also, you just need not have 2, you can have may be, you could have 3 children or you could have 4 children or may be you could have 10 children. Why you necessarily should have two children to do a search tree? Because if you had got say 3 children then the height would be log 3 to the base n. See if you have got 2 children, it is height worst case log 2 to the base n. if you have 3 then it is log 3 to the base n but then to take sorry, log n to the base 3 but to take any decision in a binary search tree, you have to make one comparison and you can move left or right. But in a 3 array tree you have to make 2 comparisons because in one comparison you will know decide exactly which one to go into, you have to make two comparisons. One is what is the value whether you will go this side or whether you will go this side or whether you will go this side. And here in a binary search tree, what did you store? You stored a node which was say 5 then all less than 5 were stored here, all greater than 5 were stored here.

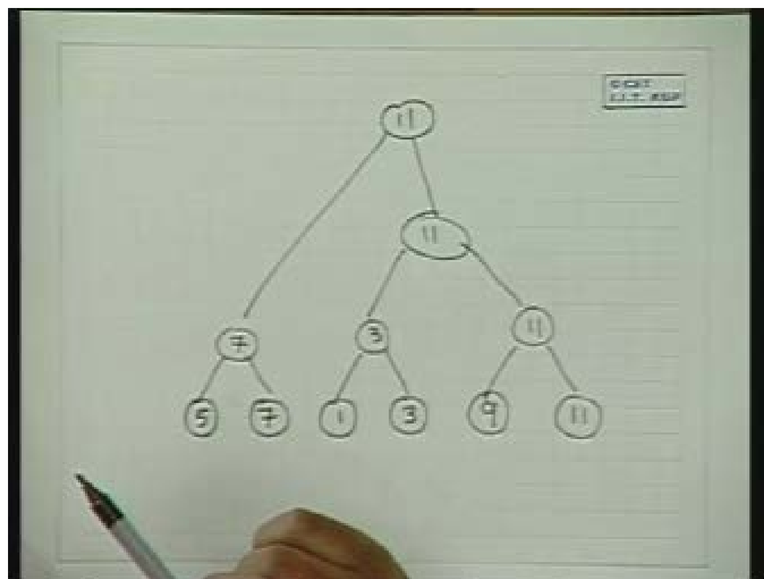
(Refer Slide Time 04:01)



But if you have got 3 then you will have to store 2 values say 7 and 9. This means that all less than 7 will be this side, greater than 7 and less than 9 will be this side and greater than 9 will be this side. Otherwise you will not be able to make a 3 array effective because you have to decide which one to go into. so you have to, your height will be less but comparison at every node will be more and both ways you will reach that you can show that more or less you are going to reach in a similar situation. The exact analysis is left for you.

Now in 2 3 trees or similar to the tournament style data structures which we did, in the binary search tree here say 2 and 7 the keys are stored at the nodes but if you recall the tournament data structures the actual node was stored at the leaf and only comparison values say for example 5 7 1 3 9 11 then you compare these two and store the largest here, compare these two and store the largest here, compare these two and store the largest here. It was like this which we stored the tournament. We could store it in an array but we stored it something like this, all right. That is the nodes were at the leaf and these contained internal comparison operations.

(Refer Slide Time 04:38)

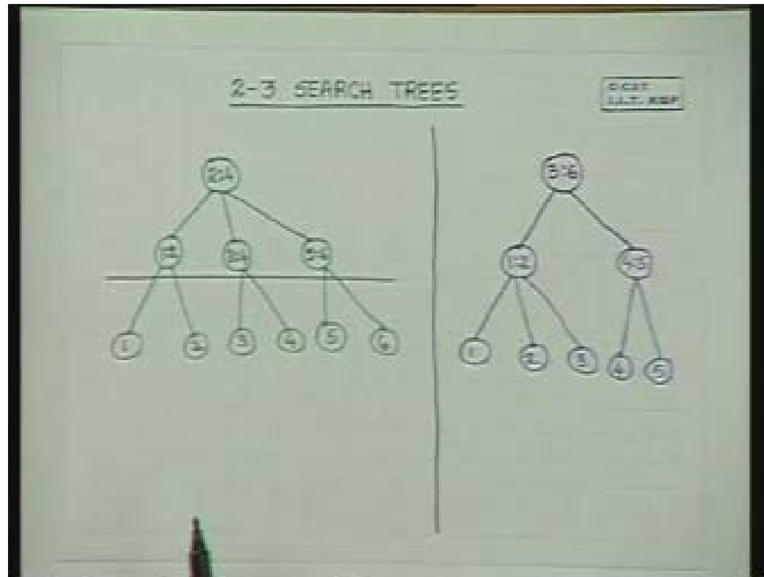


The 2 3 tree is a similar structure where the nodes are at the leaves and internally you make a comparison and store it but it is not like a tournament because at the leaves they may be unordered in a tournament. In a 2-3 tree, the leaf will also be at the ordered. We will slowly develop what is a 2-3 tree. additionally a 2 3 tree can have either 2 or 3 children, a internal comparison node, these are the leaf nodes, a comparison, these are the internal nodes, non leaf nodes.

A non-leaf node can have 2 or 3 children, all right. It cannot have 1 child or it cannot have more than 4 children. And what do these comparison nodes store? So let us have a look at two examples. These are two 2-3 search trees. The value here is 1 2 3 4 5 6 and every internal node, these are the internal non leaf nodes and these are the leaf nodes.

You need not take notes whatever material I have given you, I am following exactly that you just listen. It is given exactly in that book.

(Refer Slide Time 06:29)



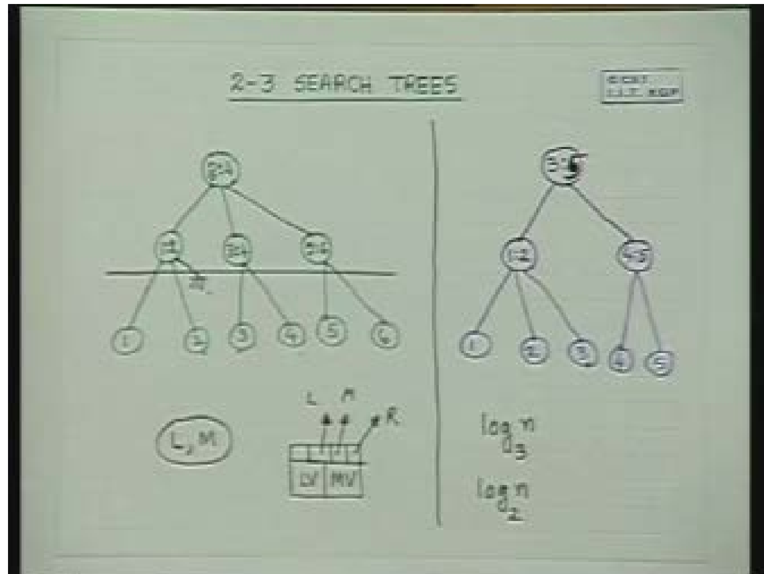
These are the non-leaf nodes and these are the leaf nodes. Now a non-leaf node can have 2 or 3 children, all of them can be 2, all of them can be 3, some can be 2, some can be 3 and they store the comparison of these elements. Now what is stored here? At every internal node, let us say for the time being we have got two values. In that classical 2 3 trees there are two values, one is called the l value, one is called the m value. The l value is the largest leaf node.

So there is a left, there is a middle, there is a right. So the structure of a node is like this. Node, node can be leaf or non leaf, all right. It can be leaf or non leaf. If it is a non leaf, it will have three pointers left, middle and right and it will have two values l value and m value, all right. The l value is the largest element along the left pointer, the m value is the largest element along the middle pointer. so at this node the largest the l value, there is no right pointer, the right pointer is null, so the largest value here is 1, the largest value in the middle pointer is 2, clear.

At this node the largest value in the left pointer is 3, the largest value of the middle pointer is 4, here it is 5, here it is 6. Here the largest value at the left pointer is 2, so 2 is l here, here it is 4 so 4 is here all right. So let us take another example. Here there are 3 children, 2 children, 2 children and this is also a 2 3 tree, here l is 1, l value is 1, m value is 2. Here l value is 4, m value is 5, here l value is 3, it is the largest on the left and m value is… Where did I get 6? 5, its 5 because it is the largest on the right, okay. And if you just reverse it like this and traverse the leaf node from left to right that is you do a traversal similar to left to right traversal, you will get all the nodes in printed order. Move left you will get the minimum, move totally right you will get the maximum and so on.

The height of this tree will be between log n to the base 3 in the best case and log n to the base 2 in the worst case.

(Refer Slide Time 10:12)

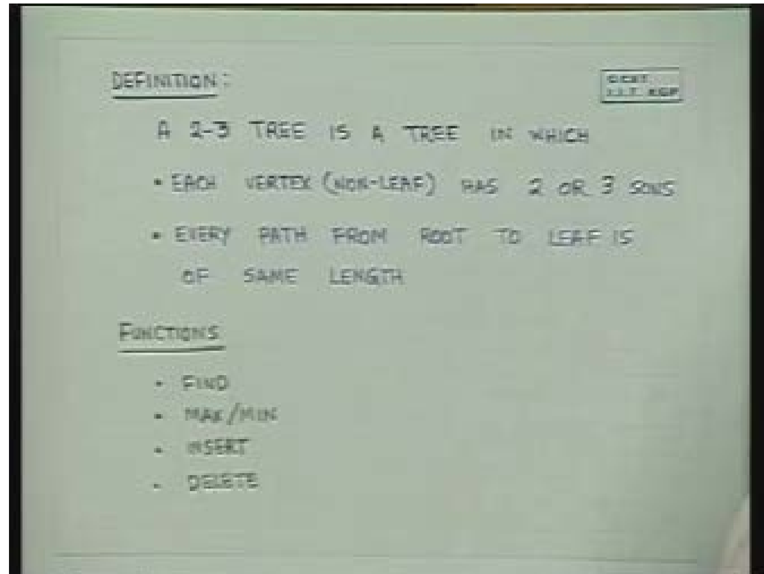


Because in a binary tree if all of them are 2, if there are n nodes the height will be log n. should I prove that? If there are 2 nodes, the height will be 1. In a binary tree if there are 4 nodes, the height will be 2, if there are 8 nodes, the height will be 3 and so on. and if you take a 3 array tree for 3 nodes height will be 1, for 9 nodes height will be 2 up to 9, 4 to 9 then 5 to 27 height will be 3 so on.

So the height of a 2-3 tree will lie between (Refer Slide Time: 11:12). so my operation to find maximum, find minimum and even the operations for moving right and moving left or insert and delete will all of them will be log n. In fact you can make them order n by maintaining the maximum and the minimum pointers which can be updated during insertion and deletion. Even in a simple binary search tree, you can make them order 1 but you will every insertion or every deletion which inserts or delete maximum element, you will have to update it.

Now let us see what we do. First let us see the definition, okay. A 2-3 tree is a tree in which each non leaf vertex has 2 or 3 sons or children what ever and every path from root to leaf is of the same length.

(Refer Slide Time 12:16)



Here note that every path from root to leaf is of the same height in a 2-3 tree unlike in a binary search tree where the height may be different, even in AVL tree there may be differences in height. Here the height like in a tournament from root to leaf will be same, will be exactly same for all the nodes okay. Is that okay?

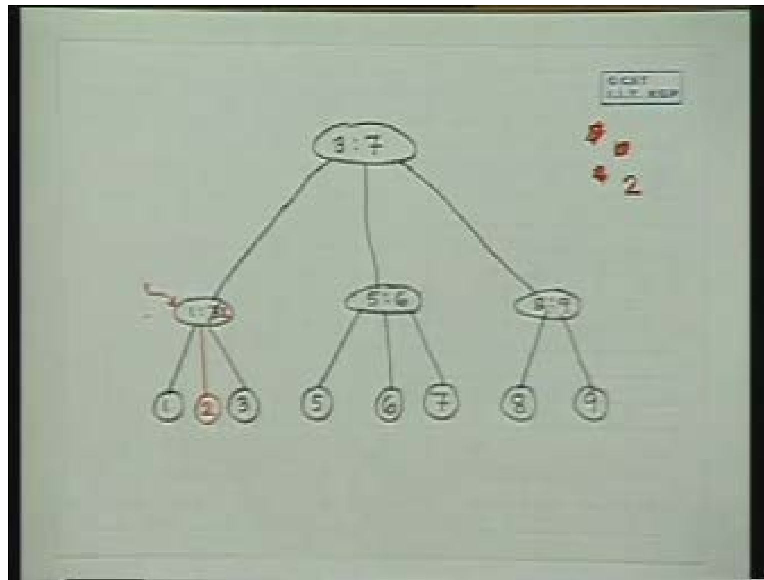
Now we will see what are the operations for find, what is the algorithm for find? Find max, min, insert, union, intersection can be easily done also. First let us see find. Suppose you are asked to find 5. What will you do? Come here find 5 at this root node, check with the l value. If this value is less than equal to the l value, you move on the left pointer else if it is less than equal to the middle value you move along the middle pointer else you move along the right pointer. See here you will move along the middle pointer, come here move along the right pointer and reach sorry, left pointer. If it is 12 say if it is 4, it will come here, you will move along the middle pointer, come here move here, once you reach a leaf node and it's not the value that you expect, you will say does not exist.

So the find operation will be the height of the tree which is log of the number of nodes because this tree, the height of all nodes will be log of n either to the base between base 2 and base 3, so it is or order log n. the maximum and minimum, obviously you move left and right you can get it that is no problem. The interesting part comes in insertion like in the earlier case, suppose we have to insert an element 2. So first thing what will you do in insert? Find, find 2 because if it exists it will not do anything or if you are maintaining duplicates, you will add the counter value. So find operation like we mentioned in the previous case will return the parent, in order to insert the find operation was returning the parent of the node where we wanted to insert.

So here it will return this and then there will be 2 or 3 children. I have taken a case where there are only 2 children, 3 is a little more complex because 3 will, if you want to insert, it will make it 4. Isn't it? So if there are 2 children there cannot be 1 child, by definition

there cannot be 1 child, there can be 2 or 3 children. In this case there are 2 children, you want to insert 2 so you will reorder the pointer and then place it in the proper position. So what would you do? You would place 2 here, make this the middle, make this the right. Anything else? Update this value, go up and see if this value needs to be updated. Here this value does not need to be updated, okay.

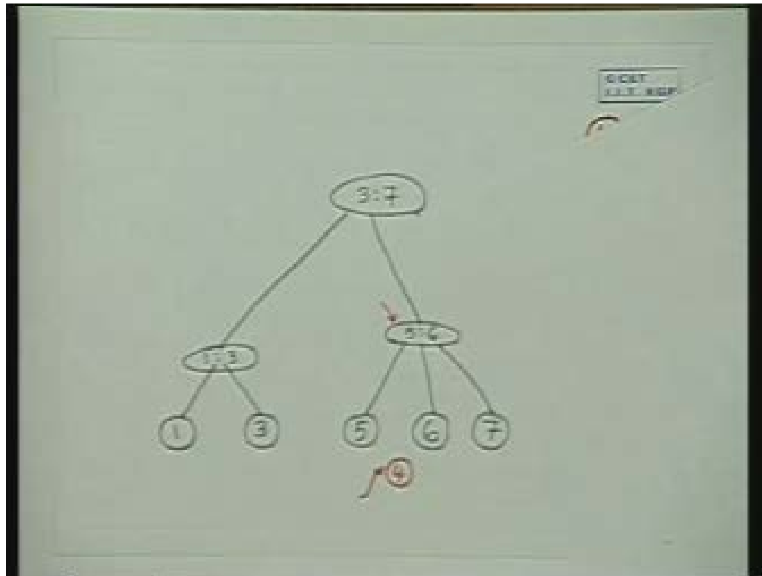
(Refer Slide Time 16:18)



So this is what will happen after insertion of 2. Suppose you want to insert 10. Then what would have happened? Move here, move here, 10. Nothing happens here, nothing happens here, you are storing maximum in the left, maximum in the middle. Suppose you store the maximum on the right also then you would have stored, you would have to update 10 but this will be sufficient.

Now suppose you want to insert 4, yes. That is the tough one. So let’s go back to our original situation and try and insert 4 because we have already inserted many. So let’s insert 4. To insert 4 we will find 4, it will come here. Isn’t it? And we will try and insert 4 and immediately you cannot put the fourth child. So to put a fourth child, you will end up in a problem. So what will you do? Okay. Suppose, this part is not there, suppose this part is not there that should simplify our problem. Suppose this is not there and you want to insert 4. So here 4 will get, what will you do, can you tell me, guess.

(Refer Slide Time 18:18)



You will have to split this node into 2 nodes, you have to make one more node, all right. So you will make one more node and here you will keep 4 and 5 and here you will keep 6 and 7. Okay, so this will be 6 7, this will be sorry, 4 will be the left and 4 will be here and 5 will be here. Now left and right let us ensure and then this will be inserted here and made the…and this value will become now 5, then what will happen.

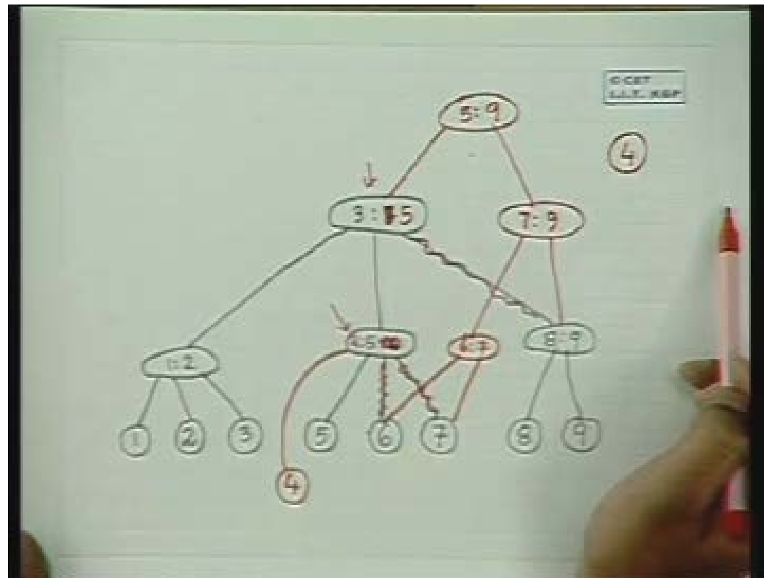
So when you go to insert, the first rule is when you go to insert a node and you see that there are only 2 children its fine. But as soon as you cross 2 children, you have to create another node and redistribute the children 2 2, you cannot distribute 1 3, you have to distribute 2 2. And then you have to insert now this node above, you have to insert this node above. Is that clear? Here it was simple, this inserting this node above was simple but it may not be always simple because when you try to insert this node then again 4 pointers may be there, then again you split that, create another new node and move up. That is what will happen in this original case.

Let us redraw this original case; I think I have it here. That is here if you want to insert 4, here you want to insert 4, okay is it visible. So again originally we will find 4 here. So to insert 4, we will split this node into 2 nodes, push this here, push this here, push this here, this will become 4: 5 6:7, this is first step. So insert 4, insert 4 in this pointer. Insert this node in this pointer, so after doing this you have created a new node. So recursively you will go from this parent to this parent and you will insert this node here. Now in the previous case there were only 2 children like this, so inserting was very simple but now to insert, you have to again split, create a new node, put this here, this will be 7 9.

Now you have ended up at the root and now you have got 2 roots, so you have to create a root. If you end up with 2 nodes and you end at the root then obviously you have to create a root and this will be now, what will be this value? This is no longer 7 now sorry, this is 5, this will be 5 9. This side actually 6 and 7 are moved this. Maintaining two values, the

programming effort may be a bit tedious, if you maintain all the 3 values then the maximum in the left, the maximum in the middle, the maximum in the right then your programming task will be easier because l value at a node is the largest value of its left side, m value at a node is the largest value of the middle child and largest value of a node is the largest value of its right child.

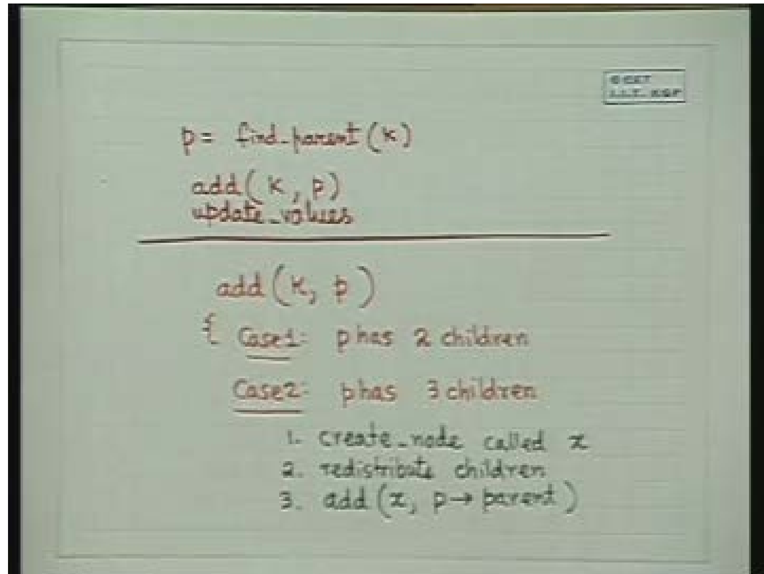
(Refer Slide Time 23:33)



So if you maintain all the 3 values, then your programming task will become easier. So what is the basic insertion routine? Insertion first will be p equal to find. Given a value, we will find the parent. The pointer to the parent right, then you will write add this k which is a pointer to this node to parent p. This is what you will do. Now come to the add routine which adds into p, which is a pointer to a node, another node. Let us assume these are two pointers okay and you are trying to add. Then what is case 1? p has 2 children, this case is easily solved, just put it there update the values. So add p and update values, obviously update values is there.

These are the l value, m value, r value updating has to be done. I am not going into that let us first understand this. And what is case 2? p has 3 children, this case is slightly more involved. What are, what is to be done in this case? Create a node, create a new node. Second is redistribute, create a node let us call it x, redistribute the children. And third can anybody tell me what is the third step? Add, this is the overall approach. Add what? x into p, parent. Isn't it? Add this to the parent f, add this new node to the parent and recursively this is the overall structure.

(Refer Slide Time 26:34)



Yeah, you have to now, maintain the parent pointer that becomes obvious that you have to be able to somehow reach that parent. So let us assume that parent pointers are there. There is no problem in creating a parent pointer, what is there. Initially make all your data structures required efficiently to move up and down then you can become more stingy on space but initially there is no need to be stingy on space. That is why even in a 2-3 my suggestion is though 2 values are kept as intermediate values, it is always advisable to keep all the 3 values.

Well, maximum at the left, maximum at the middle and the maximum at the right, it helps you to compute all these 3 values very easily and maintain all parent pointers and children pointers. So assume they are all there, you can do it in log n times without that also but in a practical scenario may be its fine both of them are okay. So this is adding and let us note that for adding, you have moved once down and you will move once up only. So you will take log n time to insert a node. So this is the basic structure of insertion okay. This is not the exact code, other than this when you complete adding, you have to check up another thing.

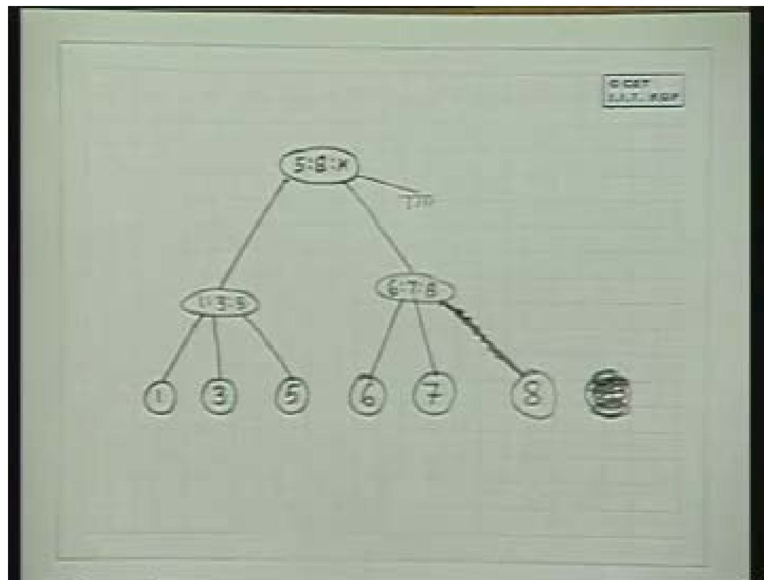
If now there are root has to be created or not. New root, if new root has to be created then you have to create a new root, that's all. Clear? So this is the overall algorithm for insertion. There is no rotation nothing here involved, its just a purposely showing this to you to give you an idea that all search trees need not be binary search trees with rotation and reflection, another that this 2-3 structure of trees specializes to other trees which are called b trees and b plus trees and they are used when you have to store in data bases where these are stored in disk, rather we are discussing all these we just stored in the memory.

In the data structure files have to be stored, huge huge records have to be stored in the disk and that point of time modified versions of these are used. They are called b trees, b

plus trees but the algorithms are very similar. So this will help you if you ever go to read a database problem in file handling in data bases, there you will see the structure of the data structures which they use are like this. The difference we are now working on the memory, all these are stored in the memory where everything is stored in the disk. So there are certain other issues but the overall philosophy is the same.

So having seen insert, max, min, is easy, let us have a quick look at delete. I have made a different tree now, forget it lets meet these 2 trees. Suppose it is this one, so this is also 2 3 tree 1 3 5 6 7 8 1. Let us store for the time being all the three values, all right. It will be easier for us to work. This is null, this value will be the maximum of this, it's very easy to compute if you have all the 3 and this doesn't make any sense because this is null so anything can be stored there. This is null, so whatever is stored there I don't care.

(Refer Slide Time 31:27)

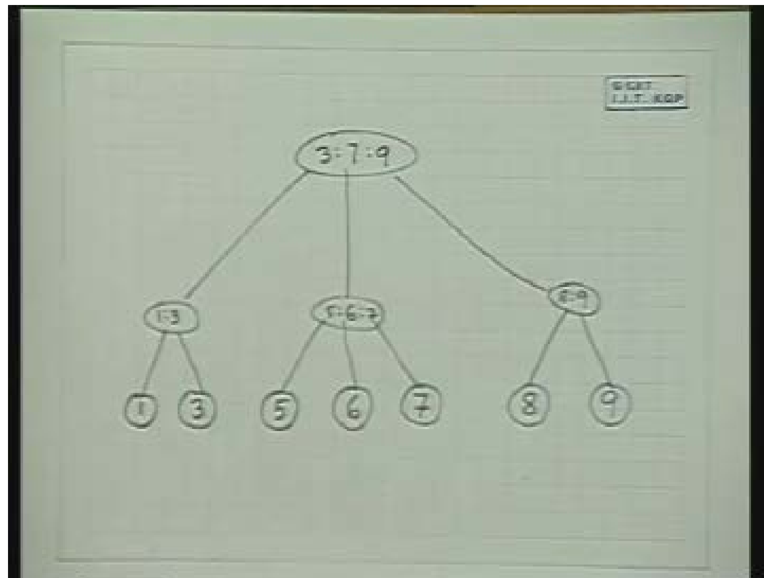


Suppose I have to delete any of these elements, it is very simple because if there are 3 children, deleting is no problem. If you have to delete 1, you just remove it, this becomes left, this becomes middle. So if you have to delete this one, this becomes if this is null then the maximum still remains 5. If you have to delete 8 but the problem will come now if you want to delete 3 or you want to delete anything, it will violate the principle. It will violate the principle of… so what will you do? Suppose you have to delete 3, so the first thing is find. In any delete operation first find it and then remove it.

So if you find it and you see there are 3 children that is case 1. In the previous case, case 1 the simple case was 2 children. Now the simple case is 3 children. If they are 3 children problem is solved but if they are 2 children, what will you do? Give me a guess. Suppose 3 is removed then what will you do? Then merge these two, the previous one was split, this one will be merge. But when you go to merge, you have to see, before you merge you have to check if this has 3 then will you merge? Just move one fellow from here. So when this is 1 you check what is called the right sibling, right brother okay. And once

you check the right brother, if the right brother has got 3 then you move the left most and make it the middle and make this the left and you just update the values. But if the right brother has 2 then you move and make this the left and remove this okay and then again remove this node from here.

(Refer Slide Time 35:21)



So once this node gets removed, this again becomes the pathological case of one just throw it away. Got it? We will take 2 examples to be more convenient. So let us delete 3. let us delete 3, if I delete 3, so delete n on the p. so this is p, this is n say k, so delete k in p. there are 2 cases, in case 1 this does not hold I am taking the bad case. Case 2, so check with this side, check with its right sibling. If it is the right most node then check with the left sibling. You may have deleted 9, so you cannot check with the right sibling this side, you have to check with the left sibling. Here there are 3, so in this case just move this fellow here, update this to 5, update this to 5, problem is solved.

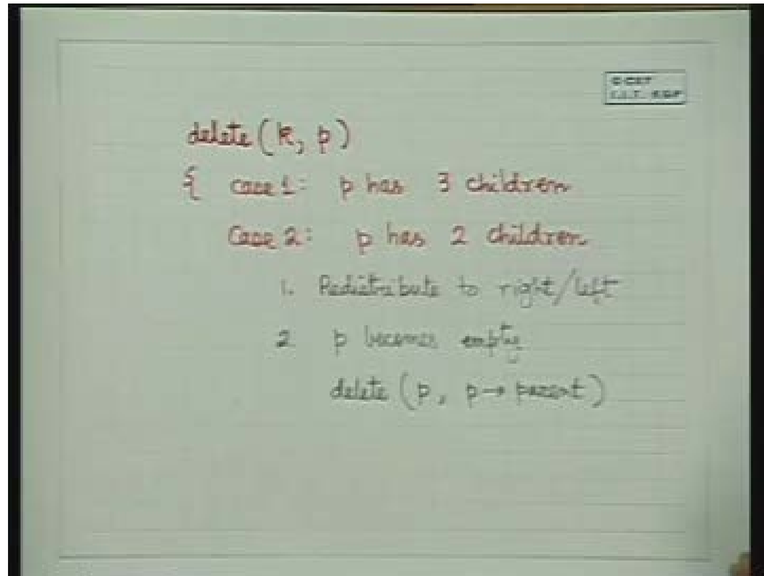
Now if you want to delete 6, fine that’s interesting. Now suppose you want to delete 6, so delete this, so you have to find 6. Delete this in this node right, look at it right brother it has got two. Since it has got two, you will push this one here. This will be 7 8 9 and then you will delete so this node has got no children. Now once it has got no children, you will delete this node in this node recursively like you have added, you will delete p in p pointer dot parent.

(Refer Slide Time 37:21)

Yeah, yeah, you could have shifted this side also, you can choose. Those cases you can choose all will work, there is no problem all of these cases will work. Similarly you can try this one and in this one, we delete anyone, you delete say 3, this will shift here. Okay. So this will shift, 1 will shift here, 3 will get deleted, this will have to get deleted. To delete this, this will shift here, this will get deleted. Now once this gets deleted, the root gets deleted, so you reduce the height by 1, like you increase the height at certain times you will reduce the height at certain times. But all these operations move once down to find and once up. So, log n plus log n which will give you log n time. So I will just try to explain this data structure which is interesting in its own way.

The delete function I think you should sit down and write it out. The delete function will be similar to the add function. Unfortunately as it always happens, the delete function is not given in the nodes. Anyway delete a node k in p, all right. So here what is case 1? p has 3 children. This matter is solved easily. case 2: p has 2 children and their first point is redistribute to right or left according to your choice and then once you redistribute, if you see p becomes empty that is no children. If p has no children then what do you do? Delete p in… it has 2 children, now if it is right… it says that p will have to become empty, no no if your right brother has got 3 children then you can just take one and push it here.

(Refer Slide Time 40:25)



If it's right brother has got 3, you just take one and push it here then p will not become empty. But if your right brother has got only 2 then p will become empty. So that way and then you delete. So this is how, this is the overall structure for delete, you can work it out. And here in a 2 3 tree all operations will take log n time. So this is to give you another flavor of a search tree and we will conclude our discussion on search trees and balanced search trees with this, just to give you an important idea because this idea is still one of the most important ideas in data structuring, specially when you have got ordered elements. That is the use of balanced search trees. So you must be, you must use it in appropriate place, you should not use a cannon to kill a mouse but you must appropriately use balanced search trees.