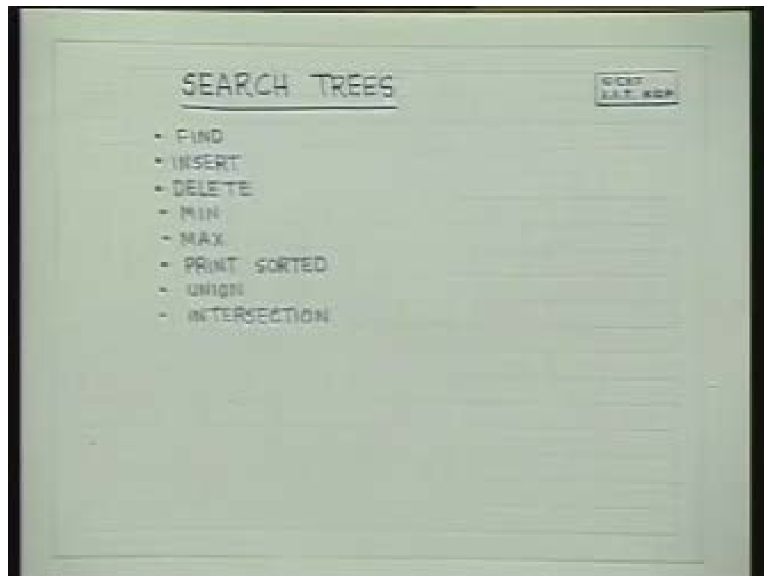
Programming and Data Structure
Dr.P.P.Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 24
Search Trees III

We have been studying data structures through manipulating sets and we studied unordered lists, ordered lists and then we began to explore the data structure called search trees which is a data structure on a set of orderable elements. That is these elements can be ordered by comparison. and our operations which we intended to look at were find an element, insert an element, delete an element, find the minimum, find the maximum, print in sorted order then union, intersection of such sets. These were the operations that we were looking at and we already found out how to write recursive routines, you can write non recursive routines also as you can easily make out. The most crucial one is the find operation then using the find, insert and delete function we saw how would we do it. The min and max were just moving to the left and moving to the right and you could find the min and max elements and then today we will quickly see what these are and try to see what are the defects of this data structure and improve it.

(Refer Slide Time 02:07)

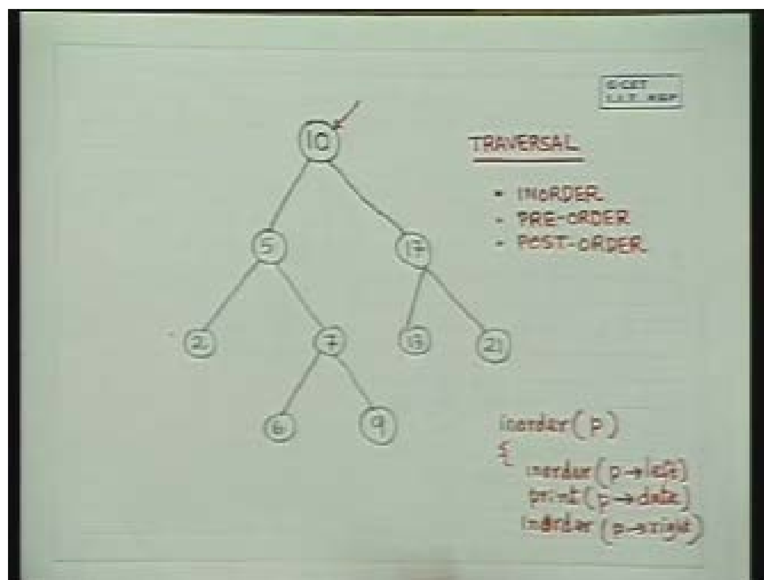


So given a set of keys like numbers or even strings which are comparable, you could set up a tree during the process of insertion and the tree was ordered this way, elements which are less than the key element at a node were at the left and which are greater than are at the right. If you have duplicates we mention that we will put in a counter and increment the counter to store how many such elements are there. And we saw how the find operation looks at the node and if it is less, if it finds the greater value equal to the node it returns this pointer otherwise it recursively searches on the left or recursively searches on the right according to the value of the variable, a value of the key variable.

Minimum maximum, minimum move totally to the left, maximum move totally to the right and you can get maximum. Deletion was a bit tricky where we saw that if we have to delete an element like say 5 then either on the left or on the right, in the left we have to find the maximum element and replace it and delete that element or move to the right and find the minimum of the right side, replace it here and delete it. We need this element recursively, so we saw the routines for doing that. And now suppose we want to print these elements in sorted order then the sequence in which we should print it is this, then this, then this, then this, then this, then this, then this 17 and then 21.

The question is how we will get it? We will get it by a simple scheme called traversal. Now there are various ways in which you can traverse a tree. The three well known mechanisms of traversal are inorder, pre order and post order and all of them vary depending on where you see the left first or the right first. So there may be two types of inorder, two types of pre order and two types of post order, sorry pre order and post order are different, inorder may be of two types.

(Refer Slide Time 06:03)

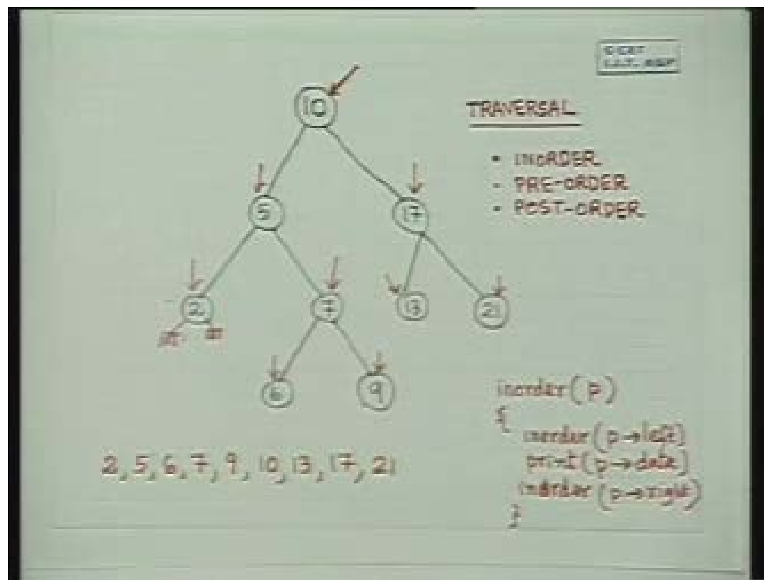


Now here inorder traversal means like this. At any node you are given a pointer to node, we have seen the node structure before. So given a pointer to a node, you do this. You first, so in order at a pointer p does this. If p is null, return null that is nothing to be done otherwise you call inorder on p left, print p data and inorder p right. What does this mean? Let us work out this. So inorder routine is this. First complete the inorder of the left then print this then do the inorder of the right. Any questions?

So let’s find out the scheme quickly here. So here, we will call it here, to do it here we will call it here, to do it here we call it here which is null. If it is null we don’t do anything, we go back. So the call here has done its left part, it will we do this, so it will print two. Then we do the right part which is null which returns, so it returns back here that means the call to this node has completed its left part, so it will print the node. So 5

and it will call it to the right. Once it calls it to the right, again this will recursively call it to the left. It will come here, this is null so it will print 6. This is null, so it will go back here and print 7 then this is here. Now here again it will print 9, go back this is completed both left and right so it will come here, this is completed its inorder right, so it will jump out of this routine and come here.

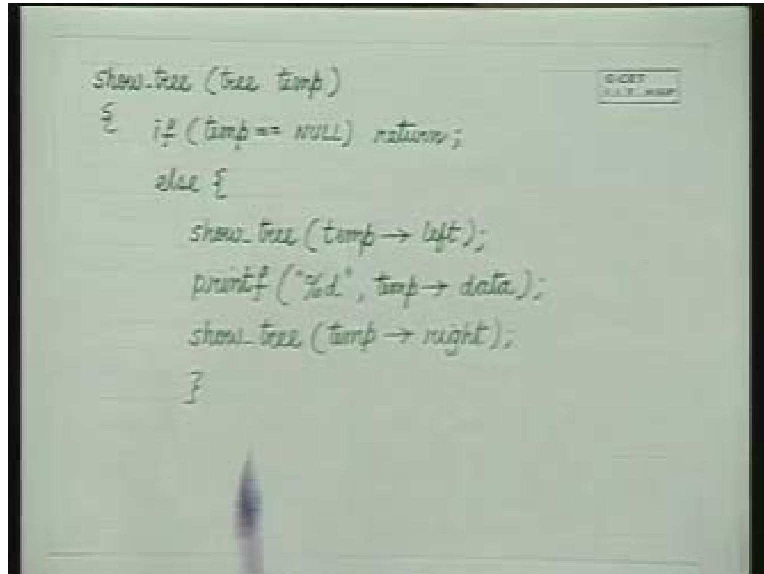
(Refer Slide Time 08:47)



Once it comes here this, the call to the root has completed its left, so it will print data and it will call it on the right. this recursively will call it on the left which will end up in printing 13, go back, print this, call it on the right, print this go back, this is completed, this is completed, this is completed. So by inorder traversal we can number the nodes also. Here we have printed the nodes that is an inorder numbering that you can do which would number this 1 2 3 4 5 6 seventh sorry, 7 8 9. So this is the sequence and if you do a inorder traversal, so 5 is called the inorder successor of two. 6 is called the inorder successor of 5. These are the successors in the inorder sequence. Trees are meant to depict other data structures as well not merely search trees or keys and we will come to examples where these things are useful, when other data representations end up in trees. But in order traversal and a simple routine like this will give you the sorted sequence in ascending order.

On the other hand if you replace the, if you read the right first and then this and then the left then you would have got in the other sequence. We have got this then this, so 21 17 13 10 9 7 6 5 2. So if you are asked to print in a sorted order, you would simply do an inorder traversal. So the inorder traversal, if somebody asks you to print in ascending order in our data structure show tree function, the tree is a pointer to the node which we have defined in the last class. If temp is equal to null, return else show tree temp left, print f temp data show tree temp right, this is sufficient to do it.

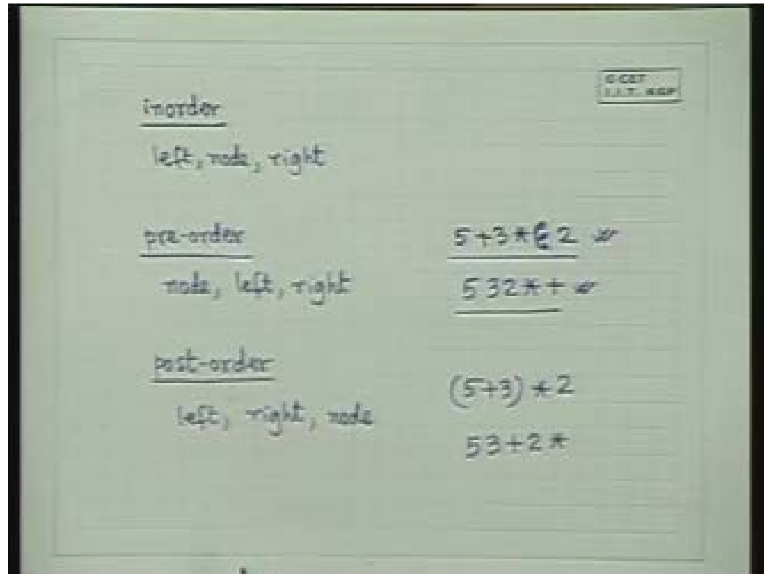
(Refer Slide Time 10:58)



And what is the time complexity, how many node it checks? It checks only the number of nodes in the tree. So it is of order n because the number of nodes in the tree, if it is n the number of edges would be n minus 1. If the number of nodes in this tree, how many nodes are there? There are 9 nodes and there will be 8 edges. In any tree of n nodes, there will be n minus 1 edges. This is not difficult to prove that, we can prove it in many ways and the simplest proof is by induction for one node there is no edge. So it is true, by basis condition is true. So the proof is very simple for one node, there are zero edges. Assume it is true for n, so for n nodes there are n minus one edges. Add one node, if you want to make it a tree and connect it up then you have to add one edge, you cannot ad two edges, it will not be a tree. So if there are n plus 1 nodes, there will be n edges so it is proved by induction, simple.

So if we do a inorder traversal then we will get it in the sorted sequence. If you want ascending or descending you have to choose whether you will do left or right first. Just for completion, what is pre order traversal? Pre order is first say inorder, inorder is left, node, right. Pre order is node, left, right. You print the node do the left, do the right and post order is left, right, node. Pre order and post order are useful in other representation of trees not in such search trees because here they will not give you exactly what you are requiring but if you remember, do you remember post fix sequence, a number. Suppose the expression 5 plus 3 into say 2, this can be written in post fix notation as 5 3 2 star plus. Have you heard of this post fix, pre fix and in fix notation of numbers, of expressions? And any expression, the advantage of, what is the advantage of post fix over normal in fix? Suppose in in fix, you write (5 plus 3) into 2 or x plus y into z. In post fix it will be 5 3 plus 2 star, the advantage of post fix is no brackets are required.

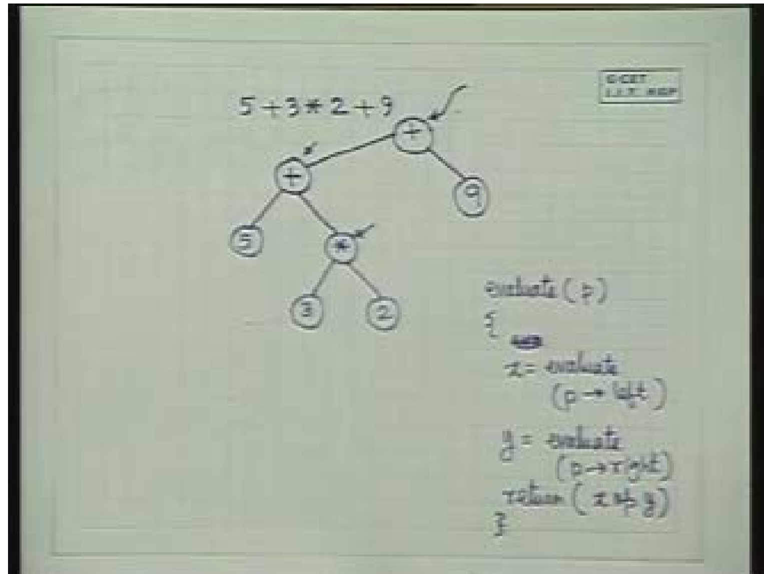
(Refer Slide Time 14:49)



The pre fix notation in the computer lot of expressions are stored as post fix notations. So trees are not only meant for binary search trees. Expression trees are very famous data structure for storing trees. for example if you have a expression, 5 plus 3 into 2 plus 9, then an expression tree for this is usually a node, an expression tree would look like this. This is a different data structure, I am just trying to point out that such expressions are usually stored in the form of trees like this. So this says that to get the value of this expression, first you have to get a value here then get a value here and then get the value here.

So to evaluate a node here, the evaluate function at a node p I am not, if p is null you return null otherwise you evaluate get x which is the evaluate p left, y is evaluate p right and then return x and this operation op, this may be plus minus multiply divide whatever op y. This is the recursive condition you evaluate left, evaluate right and get op. this is how you evaluate an expression and the base conditions are if this is not an operator, if it is a data you just return its value. This node may be of type operation or data, if it is a data you return its value otherwise you do this.

(Refer Slide Time 17:11)



Now, given such an expression tree which is a common data structure, if you store the expression, if you print out the inorder expression tree and for every node, every time you print it out, you print out the brackets properly. That is in order would be first left side. So here you come, again first left side so you will print 5 then node then this side, so to do this side, so whenever you are at a node you print a bracket.

Okay? So you start printing your right bracket and then come here, so here you printed your right bracket let's start again, you print a right bracket. First thing you do is print a right bracket and the last thing you do is print a left bracket, sorry print a left bracket first and the last thing you do is print a right bracket. So and then do the traversal, so first you print a left bracket then move left. Here this is not null, print a left bracket move left, here this is a data 5, print a right bracket move back, print the operation, do this side. When you come here print a right bracket, move left, print 3 close it come here, print this node then come here, come back when you finish, print a right bracket.

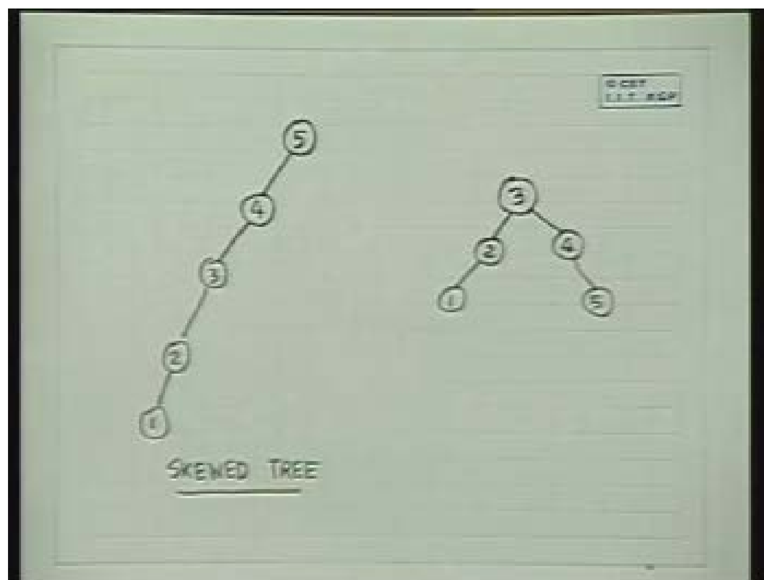
Okay, I should have printed another bracket out here then come back print this, come back print this, come back print this, you will get the normal inorder expression that we write infix notation. If you do a post order traversal that is left right node then what would be the sequence? Left, left, left so this is first 5, right right then you will come here left 3 then again here 2 then come back, star and comeback plus, come this side right 9, come this side plus, you will get the postfix notation. What does this mean? 5 3 2 so these two will be done first, so this two will be done so you will get 6 then this and this and this will be done, post fix evaluation and then what ever you get the 5 plus 6 is 11, 11 plus 9 will give you 20. This is how you will get postfix and if you do preorder, you will get the prefix notation.

So this is another use of trees as a data structure not merely as such but let us come back to our concept of search trees. And here by doing an inorder traversal we can print the

nodes in sorted order. Now once we can print the nodes in sorted order and we know which is the inorder successor of a node. Then trying to do a union operation is not difficult because if you recall when we had ordered set of numbers 5 7 12 15 3 4 19 then we said that a union algorithm will be a modified version of the merge algorithm of merge sort, you compare this two and the smaller one we put it in.

Similarly here if you have to do a union of two trees, we just move according to inorder successors and you will be able to get the union very easily because the inorder successor will give you this sequence. So we can use an algorithm which is similar to that to get an algorithm for union as well as intersection. So search trees have got lot of potential for solving a number of search based problems. that is find, insert, delete, min, max, print, sorted, union, intersection these will be done the potential of using search trees is better than of using ordered lists. Obviously in the worst case why should it be better? Because all these operations union and intersection will take order n times where n is the total number of nodes in the two sets, anyway.

(Refer Slide Time 24:07)

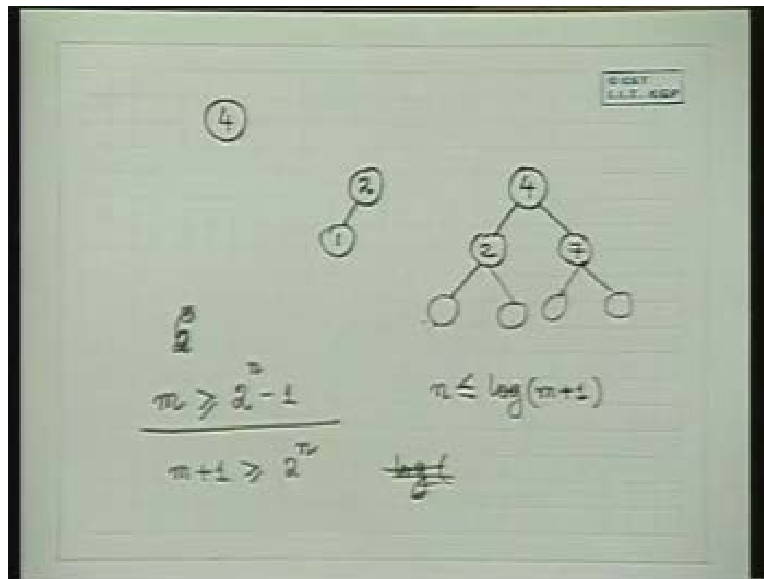


But sorted will also take time proportional to the number of nodes but these find, insert, delete, min, max are going to take time which is proportional to the length of the longest path in the search tree. And the length of the longest path in the search tree in the best case and the worst case what it can be? Suppose you have a search tree, in the worst case it can be a skewed tree and a skewed tree has got its, worst length of the longest path is the number of nodes but in the best case this tree would well have been in a better situation, this tree could have been like this. Now 1 2 3 4 5 nodes, given 5 nodes can we get a tree whose length of the longest path is less than 2.

Let us see, for one node let us see what is the best situation that we can get? Okay. The worst we have found out but what is the best? For one node it can be like this, for 2 nodes it cannot be 0, it has to be at least one, so for two nodes may be it will be 1.

Now for 3 nodes the best case can be one, so for 3 nodes say 4 2 and 7 for 4 nodes it has to be at least 2. So for 4 nodes it has to be 2 and how long it will remain up to how many nodes? Up to 7 nodes. See 4, 5th node see 3 4 5 6 7 up to 7 nodes it will remain two, from the 8th node onwards it will again have to be 3, you cannot avoid it. So from 8 upto how many it will be 3? 15th node. So now you are getting the pattern? The pattern is that the best case is for up to 2 to the power, suppose you have got m, take the largest m such that m is greater than 2 to the power greater than or equal to, take the smallest m such that m is greater than or equal to 2 to the power n minus 1. This is going to be the best possible that you are going to get. Isn't it?

(Refer Slide Time 24:07)



So what is m? m plus 1 must be greater than equal to 2 to the power n. so log of, so what can you say about n and m plus 1? n is equal to, n must be less than equal to, so the best case height will always be bound by log m plus 1. So now let us come back to our operations. so in the best case normally find in an ordered list would take order n time, here it will take order log n time, insert should take order log n time, delete should take order log n time, min and max all log n time. This is best case scenario, this is not the worst case scenario. So in an ordered list it would take n, n, n, 1 constant time, min and max constant time, this is n, n, n. here we are trying to say we will get, we may get it like this.

(Refer Slide Time 29:49)

SEARCH TREES

- FIND	n	log n	
- INSERT	n	log n	
- DELETE	n	log n	
- MIN	1	log n	
- MAX	1	log n	
- PRINT SORTED	n	n	
- UNION	n	n	
- INTERSECTION	n	n	

WORST CASE HEIGHT IS $O(\log n)$

BALANCED SEARCH TREES

And if these operations are equal, we have reduced the worst case here in many problems to log n but we have increased some of them that depends on the frequency in which this operations come. But the question is still this is the best case. So can you modify the data structure, so as to ensure that we will achieve this. Search trees were during insertion and deletion, you try to make sure that the worst case height is of order log n. It may be twice log of n but order log of n, in the worst case this we will try to ensure during every insertion, deletion. So we will have to modify our insertion, deletion in some intelligent way.

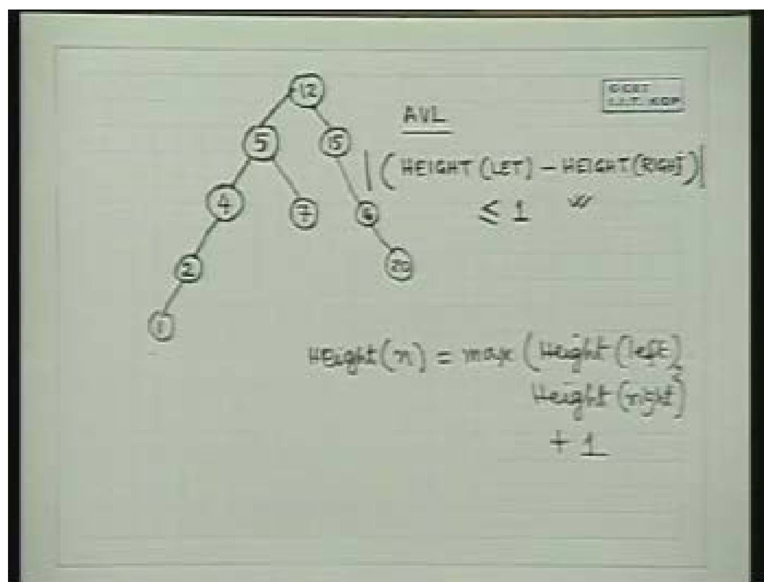
If we could do this then we would have achieved something important. Such search trees are called balanced search trees. And there are various such balanced search trees. There is not a single type of balanced search tree, there are various types of balanced search trees. We will hear of several types of balanced search trees data structures. The earliest and one of the most original balanced search trees was called the AVL trees. It is named after the people who discovered or invented this data structure Adelson Veiskii and Landis. That is why it is called the AVL tree. There are several other search trees among binary, this is a binary search tree among, binary search trees the most popular balanced search tree nowadays is called, it is a modified version of a AVL tree, its called a red black tree. And there are other balanced search trees which are not binary trees. Okay? Such amongst them you have, the most famous among them is the 2-3 tree.

So these are some names, so we must now understand what they are actually supposed to do. In any balanced search tree, the insert the find operation, suppose it's a binary search tree okay the find operation remains the same, there is no change in the find operation in AVL tree or in a red black tree. In a AVL tree specially which is just a binary search tree which is balanced there the find operation does not change but how does the insertion operation change? So as to ensure that the tree is always balanced, without going into the algorithm I will just take one example and show you what happens.

Suppose 5 is inserted and then after that suppose 4 is inserted, so 4 will be inserted by the normal insertion algorithm like this then next say 7 is inserted then 7 would be inserted here and every time insertion takes place, two things are completed. The height of the left tree and the height of the right tree. Okay? And in the AVL tree, the check is height of left minus height of right. The absolute value that is the difference either they will be equal or they will differ by at most 1, they will never differ by 2. That is you are not expected to have a tree.

Here this is a valid AVL tree but this is not a valid AVL tree because the height, this is true at every node, this is true at this node the height of this tree must not be violated. Now suppose here, height here is height of this is 1 2 3 height of this is 1 2 3, this does not mean that this tree is balanced because this AVL condition must hold for every node, all right.

(Refer Slide Time 35:27)



So the AVL tree says that at every node, the height of the left tree and the height of the right tree must differ by at most one. You can prove that if this condition holds at every node then in the worst case, the worst case length of the height will be order log n. How you will prove it? I will leave it to you to prove to see why it happens like this, if this condition is true. So once it detects, how does it detect they are maintaining a height value at every node and once the height value of the node is updated, the height value of its parent can be updated, the height of the parent is maximum of the height of its children plus one.

The height of a node can be recursively computed is maximum of height of left comma height of right plus 1. Isn't it? The height of this node is what? the height of this, height of this, take the maximum plus 1, here it is 3, here it is 2, so this will be 4. So using this height value at every node, the AVL tree does what is called the concept of balancing.

Quickly I will show you, suppose 12 is inserted then 5 then 4, this violates the height concept because the height of the right is 0 and this is 2. Immediately it is given a rotate right, this rotate right means move this as the right child of this and move this up. So it converts this to 5, 12 it does a rotation. If it is skewed on the other side, it will do a rotate left and this once it rotates it right and then it will continue to balance the right side. So balancing, without going to details just like to point out that balancing during insertion in an AVL tree can also be done in log n times. So during insertion the total process of insertion and balancing will be done in log n times.

Similarly during deletion after deletion if it finds that the height value changes again that rotation will be given but in the reverse direction. So the concept of rotation is that both in an AVL tree and in a red black tree to give, to maintain the balance of a tree. I will not go into the details of a red black tree or AVL tree which is available in the text books because it is a binary search tree and you can have a look at it from any book, just to give you a flavor that search trees need not be binary trees, we will look at another balanced search tree called 2-3 trees in the next class.