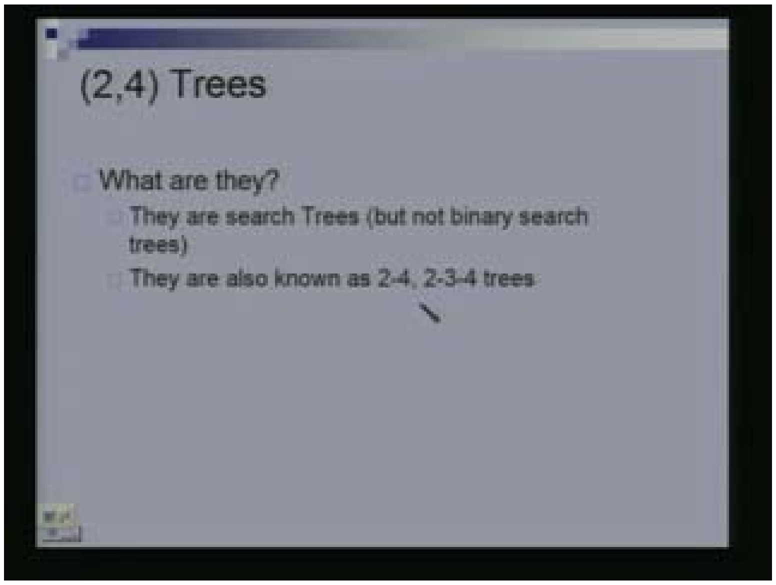
Data Structures and Algorithms
Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture – 13
(2, 4) Trees

In today's class we are going to be talking about 2-4 trees. This is another way of representing a dictionary. So we are going to see the operation of insert, search and delete on this data structure and we are going to have the same kind of performance guarantees as the case in AVL trees. But in later classes we are going to see how this data structure is useful. So today I will just begin with this.

(Refer Slide Time: 02:10)

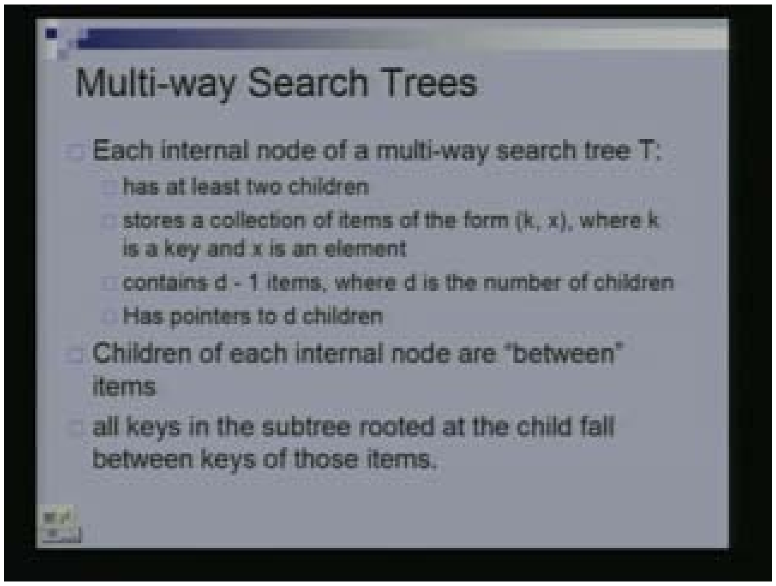


(2,4) Trees

- What are they?
  - They are search Trees (but not binary search trees)
  - They are also known as 2-4, 2-3-4 trees

What are 2-4 trees? They are search trees, they are a kind of search trees but they are not binary search trees. So recall in a binary search tree what was happening? The tree was a binary tree with each node at most 2 children. So this not going to be a binary tree. That is a first point. Nodes can have more than 2 children now. So these 2-4 trees are also called 2-3-4 trees. I will tell you what this really means. So 2-3-4 actually refers to the number of children and node can have. So a node can have either 2, 3 or 4 children. Such trees in which a node can have many children but satisfy a certain kind of search properties are called multi-way search trees.

(Refer Slide Time: 04:43)



Multi-way Search Trees

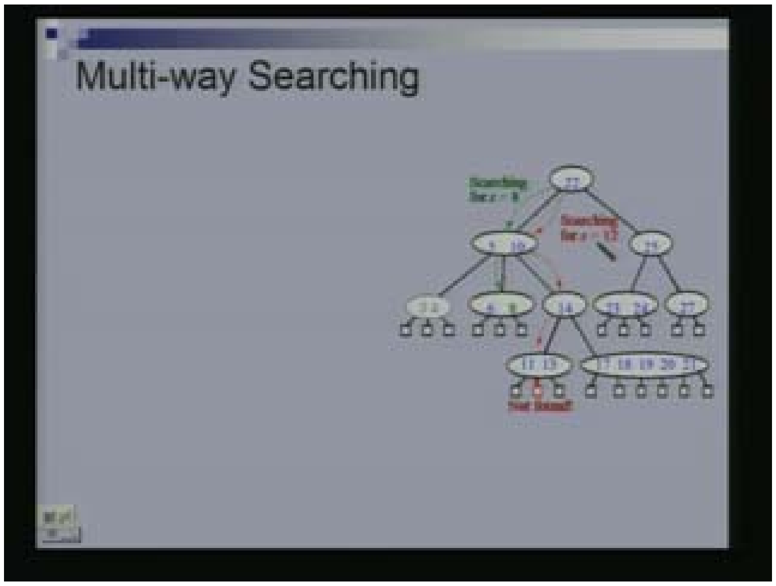
- Each internal node of a multi-way search tree T:
  - has at least two children
  - stores a collection of items of the form (k, x), where k is a key and x is an element
  - contains d - 1 items, where d is the number of children
  - Has pointers to d children
- Children of each internal node are "between" items
- all keys in the subtree rooted at the child fall between keys of those items.

So each internal node of a multi-way search tree has at least two children. It will have at least two which means it could have more than two children. Any number of children more than two. Each node of a tree also stores a collection of items of the form (key, element). In the binary search tree, each node was storing one key and the element there was let's say a reference to the element or the element itself could be stored there. If the key was a student entry number, then the student record associated with the key could also be stored in the node itself. So in the similar way, we have that in the multi-way search tree you will have each node containing a pairs of this kind that is (key, element). And how many pairs there could be? It is more than one. In the binary search tree there is only one such pair in each node and in a multi-way search tree there could be more than one.

In particular there could be d-1 such pairs or items, where d is the number of children that particular node has. So we are just generalizing the binary search trees. In the binary search trees each node has two children. Each node could have two children and then there is only one key that is kept in the node. Because that key helps us to determine whether we should go left or right. Similarly here we have d children, if d is the number of children then you really need to know in the search process whether you should go to the first child, second child, third child, fourth child and so on. So you will have d-1 different keys sitting in the node to help you determine that. I will soon show you an example and that will be clear.

So this is an example of multi-way search tree (Refer Slide Time: 04:50). As you can see this node has two children. This has 3 children, this node has 2 children and this actually has 6 children. How many keys are there in a node? The number of keys in a node is one less than the number of children that node has. And why is that? So for instance this node (Refer Slide Time: 05:20) has three children and you need two keys in the node. The keys in the node determine what set of keys the various sub trees are going to have.

(Refer Slide Time: 04:43)



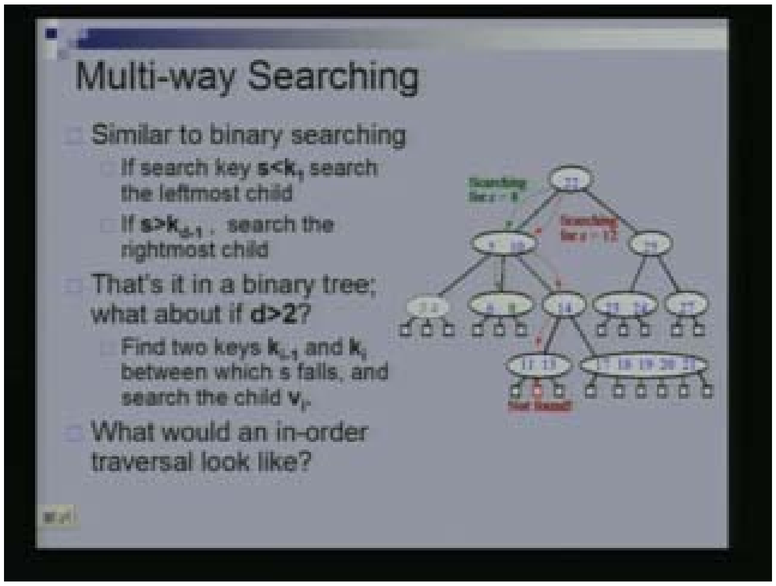
Multi-way Searching

So what I am trying to say here is, this is key 22. So this is also in the left sub tree. So everything in this left sub tree here is less than 22 and everything here is more than 22. Everything here is more than 22 and everything here is less than 22 (Refer Slide Time: 06:00). If you look at this key everything to the left of this is less than 5, so in the right sub tree we now have 3 children. So in this first sub tree everything would be less than 5. In the last sub tree everything will be more than 10 and those in between 5 and 10 would lie in this middle sub tree. So that is a concept. Now you understand why you need d-1 keys if you have d children. So everything less than the first key would be in the first sub tree. For that you have to follow the first child. Everything between the first key and the second key you will have to follow in the second child and so on.

With that let me go back to the previous slide (Refer Slide Time: 04:43). So the children of each internal node are between the items. This is what I mean by between in code. So you have a certain node it has various keys or items. If you look at two consecutive keys then all the elements or all the items which have key value between the consecutive pairs would be in one sub tree. For that you will have to follow one child. So let’s get back to this. This is an example of a multi-way search tree. And how do you search in such a tree. Searching is similar to the binary search procedure as you did in the binary search tree.

So suppose we are searching for 8. You come down here, compare 8 with 22 so 8 is less. So you go here, now you will have to find, so 8 is not less than 5 and 8 is not more than 10. But 8 lies between 5 and 10. So you will follow this and then you will find that 8 is sitting here. So it’s a successful search. So when you are searching for a key s you will compare it with k_1. k_1 is lets say the very first key in that node and k lets say k_{d-1} is the last key in that node. So you compare it with the very first key if it is less then, that means you have go to the left most sub tree. If it is more then k_{d-1} then you have to go to the right most sub tree.

(Refer Slide Time: 09:25)



Multi-way Searching

- Similar to binary searching
  - If search key $s<k_1$ search the leftmost child
  - If $s>k_{d-1}$, search the rightmost child
- That's it in a binary tree; what about if **d>2**?
  - Find two keys k_{i-1} and k_i between which s falls, and search the child v_i.
- What would an in-order traversal look like?

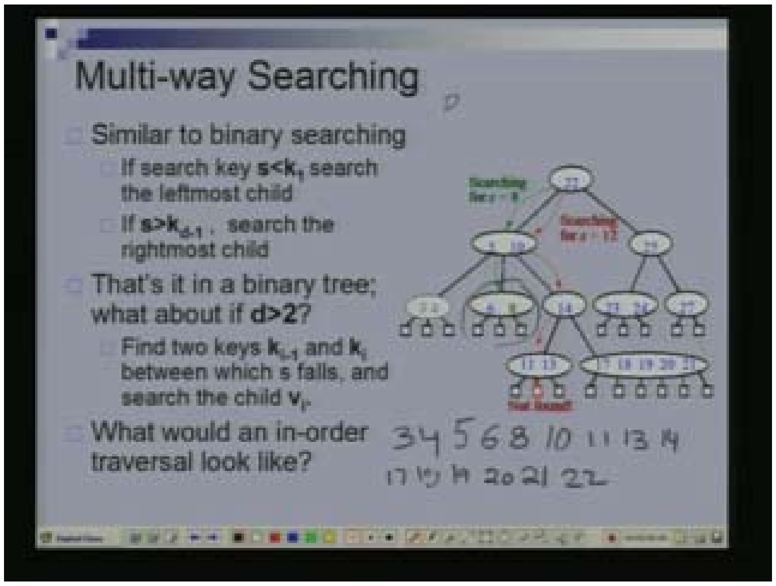
Searching for s = 8
Searching for s = 12
22
5 10
25
3 4
6 8
14
23 24
27
11 13
17 18 19 20 21
Not found!

So when you are searching for the node, for instance when we are searching here for 8, we came down, we went left because 8 is less than 22. Then 8 lies between 5 and 10. So we came down here and then we found 8 here. So when we are searching for 22, we came down the sequence of steps and we found that 12 was not there in the tree. So in particular when you are at a node, you have to determine that the key that you are searching for lies between which 2 keys and once you determine that you will follow the appropriate child. At the two extremities you will check whether it is less than the first key or it is larger than the last key. In which case you would follow either the left most child or right most child. So it is as simple as that.

So what would an in order traversal in the tree would look like? That was the question we were at. So first what is the in order traversal in a tree? We recall in order traversal says left, then you print the data of the node and then you go right. But now there is no left and right, because a node can have many children. So what does an in order traversal here mean? So first go the left most then print the key, then go to the next child then print the key, then go to the third child then print the key, then go to the next child and so on. That would correspond to an in order traversal. So for instance here if were to do an in order traversal, what would I do?

I would come down here, first go left. So first I will do an in order traversal on this part of the tree. Which means that I first come in here, I first go left I will do an in order traversal here, which means I come in here I first go left but there is nothing here. So then I print the key, that is 3. Then I go to the middle child, nothing there so then I print the next key 4. Then I go to the right child, nothing there. So that finishes the in order traversal on this node.

(Refer Slide Time: 11:45)



Multi-way Searching

- Similar to binary searching
  - If search key $s<k_1$ search the leftmost child
  - If $s>k_{d-1}$, search the rightmost child
- That's it in a binary tree; what about if $d>2$?
  - Find two keys k_{i-1} and k_i between which s falls, and search the child v_i.
- What would an in-order traversal look like?

3 4 5 6 8 10 11 13 14

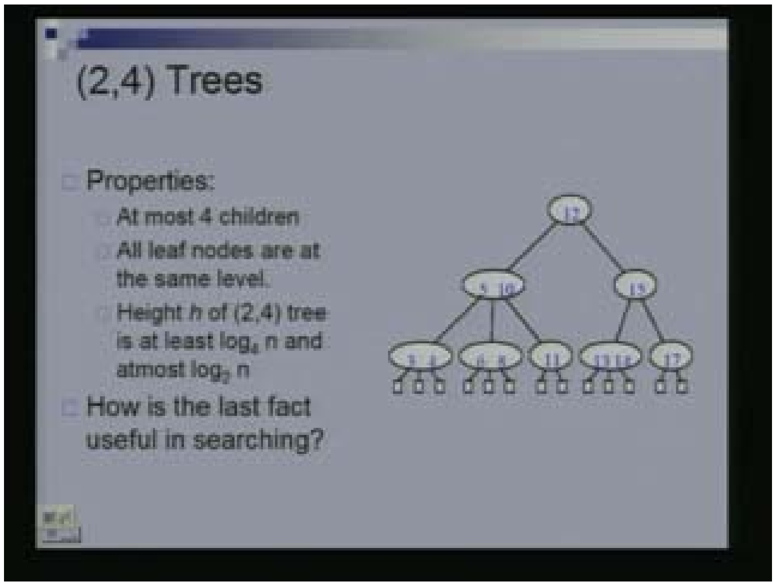
17 18 19 20 21 22

Having finished the in order traversal on this node, I go back to the parent. And then I will print this key, because first I went left then I print the key which means I print 5. Then having printed this key I will now do an in order traversal of this sub tree (Refer Slide Time: 11:15). So when I am doing in order traversal of this I will get 6 and 8, having finished that I go back and print this key now which gives me 10. And then I will do an in order traversal of this right sub tree. And that would give me 11 first, 13, 14 and then all of these. So 17, 18, 19, 20, 21 now I finish the in order traversal of this entire thing. So I print the key 22 and then I go right. So as we can see, you will get the keys in sorted order. That is also easy to prove. Why? Because in an in order traversal I will first print out all these keys and only then will I print out this key.

So which means that in the order of printing all the keys which are less than this key will appear before and all the keys which have more than this key will appear after and this is true for every key. So which means what you get is a sorted order. Yes, 8 could have more than other children also. For instance, I could have something here let's say 5.5. The 5.5 would be a valid node here? Instead of this I could have just one node with 5.5 here. That we could have organized in a different manner. But 5.5 is a valid node, there could be more nodes here.

So now let us understand what 2-4 trees are. So 2-4 tree is something like this. What are the properties? Each node has at most 4 children. So first it is a multi way search tree. Multi way search tree which means every node has at least 2 children. Now we are saying each node has either 2, 3 or 4 children. That is why it is called a 2-4 tree or 2-3-4 tree. Each node has at most 2, 3 or 4 children. The second important property is that all the leaf nodes are at the same level. So the leaf node here are this, just forget this square boxes for now. So these are the leaf nodes and they are at the same level.

(Refer Slide Time: 16:28)


12
5 10
15
3 4
6 8
11
13 14
17

(2,4) Trees

- Properties:
  - At most 4 children
  - All leaf nodes are at the same level.
  - Height *h* of (2,4) tree is at least $\log_4 n$ and atmost $\log_2 n$
- How is the last fact useful in searching?

They are all at level, suppose we are numbering level 0, 1, 2 again, so they are at level 2. These are the only 2 properties of a 2-4 tree. Of course it is multi-way search tree, so a 2-4 tree is a multi way search tree with these 2 additional properties. Search tree will have a property that, everything which is less than this key is going to be in the left and everything that is more is going to be in the right. This is an example of a 2-4 tree, as you can see this node has 3 children and this has 2. There is no node with 4 children but you could also have a node with 4 children in it.

What is the height of a 2-4 tree? Why should the height of the tree be at least $\log_4 n$ and at most $\log_2 n$? What is the worst case? When would the height of the tree be maximum? It is when everyone has 2 children. In that case everyone has 2 children and all the leafs are at the last level. Then it is exactly a complete binary tree. And in complete binary tree we argued that the height is $\log_2 n$, there was plus one, minus one some where forget where it was, but it is some thing like that. That is a setting when the tree height is maximum. The tree height is minimum when every node has 4 children in it. Because then the nodes are closer to the root, you will have 4 and then 16 at the next level and then 16 times 4 that is 64 at the next level and so on.

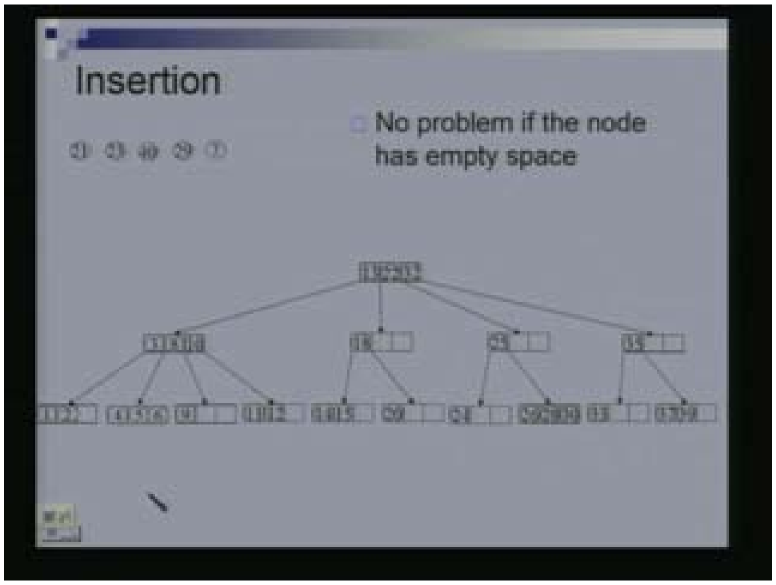
Once again if we do the same analysis you will find that the height of this tree is $\log_4 n$. So height of the 2-4 tree on n nodes always lies between these two quantities. It is either $\log_2 n$, it lies between $\log_2 n$ and $\log_4 n$. $\log_4 n$ is essentially half of $\log_2 n$. Basically the height of the 2-4 tree lies between half of log n and log n. How much time does it take to search in a 2-4 tree then? Why log n? How do we search in a 2-4 tree? It is a multi way search tree. So if I am searching for a particular key lets say suppose what do I want to search? I want to search for 11 lets say. I came here with 11, where would I go? Compare 11 with 12, I come here (Refer Slide Time: 17:10). I am comparing 11 with 10, so I go right and then I find 11 here. So I found 11.

So how much time does it take for me to search in a 2-4 tree? Height of the tree, is it something more that I need to do? It is correct, it is order height of the tree. I have to compare with in each node. Because when I am searching for 11, I have to essentially compare against, how many keys there could be in a node? A node we said has how many children? 4. If it has 4 children how many keys would it have? 3. The maximum number of keys therefore is 3. If it has 2 children how many keys do I require? 1. So a node has either 1, 2 or 3 keys. So when I search for the key and I come with key, then I have to compare it with this key, this key and this key (Refer Slide Time: 18:16). So I might require 3 comparisons in all to determine which particular branch to take out. So I might require 3 comparisons.

So the time is 3 comparisons with in a node, times log n because that is log n is a number of node I would be visiting. Order log n. So order log n is correct but you have to be careful about this. Within each node you require more than one comparison. In a binary search tree you required only one comparison but now you could require up to 3 comparisons. Why 3 log n? He is asking me why did I say 3 log n. When I am searching, I start from here (Refer Slide Time: 19:06) start at the root and then whatever key I have I compare it with the keys in a node. Here this node has only one key but it could have 3 keys in it. Then I have to compare against each of those 3 keys to determine which particular branch to take out of that node. If it has 3 keys then there are 4 different branches, which should I take? To determine that I need to make 3 comparisons.

Let us look at insertion in a 2-4 tree. I have this largest example that I am going to be using to show you the process. Is this a 2-4 tree? This has 4 children, this has 4, this has 2 children, this has 2 children and this last one also has 2 children (Refer Slide Time: 19:56). I have shown the node with 3 locations in it. So each node will have space for 3 keys and 4 pointers.

(Refer Slide Time: 19:43)



Insertion

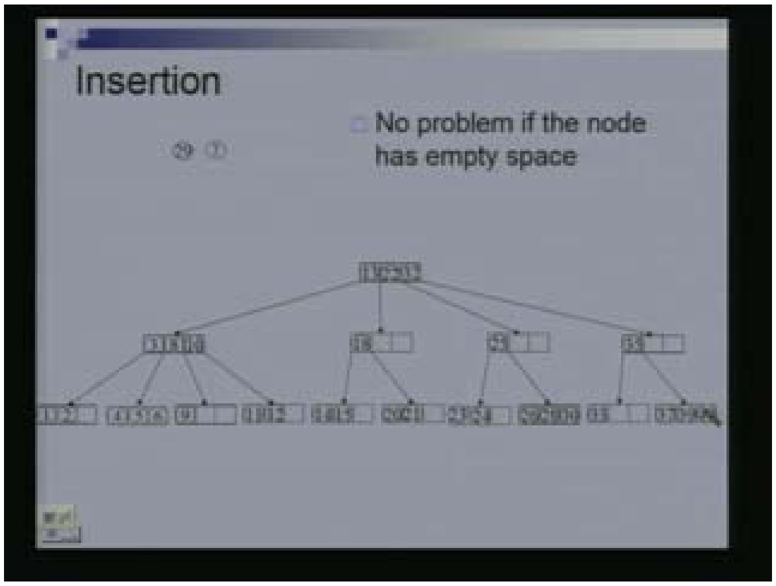
- No problem if the node has empty space

So it has only one key but I have shown each node having space for 3. So the first element I am going to insert is 21. How do we insert? We insert just as in the case of the binary search tree. First we will search and wherever our search terminates, if we found that element then it would say that it already exist. You will not insert then but wherever the search terminates we would insert the element there. I am trying to insert 21. So 21, I come and compare. Here 21 lies between 13 and 22 which means I am going to take this branch out. So take this branch out, I compare it against 18. It is larger than 18 so I am going to take this branch out (Refer Slide Time: 21:00). So take this and it goes and sits in that particular node. Why does it go and sit in this node? Why did not I compare with 20 and say that let me go down further. This is a leaf node. I could also have said it, I compare it with 20 then I try to go right but right node is empty. The right pointer is a null pointer because it is not going down any further. So I know that this is the place where I have to insert and this is empty and there is space here so I just put it in.

We would not put it next to 18. We would continue till we can not go any further. This is what happens in the binary search tree. (Hindi conversation) you keep comparing till you hit a null pointer and then you put it there. So till we hit a null pointer. We compared 21 with 20, let's say we were trying to go right but this is a null pointer and so we put the node here. Now you are wondering how I am going to use this space. We will see how we are going to use this space. If this was already filled (Refer Slide Time: 22:22) (Hindi conversation) you will have to wait till the next slide. So if there is empty space no problem you can do the insertion.

Let's say now we try to insert 23. So 23 lies between 22 and 32. We are going to take this link out. We took this link out, 23 is less than 25 so we come down here. In a node we will try to keep the keys in a sorted order because only then. So 23 should come at this place. What should I do? Move 24 to the right and 23 will come at its place. Insertion actually happens at the very last node that is at the leaf nodes. The other way I could think of it is 24 was here, I compare 23 with it, I tried to go left that is null pointer. So that means I have to insert at that node itself. We are trying to insert 40. There should be no problem with 40. The 40 is more than 32 so I go right, I come here 40 is more than 35 so again go right and there is space here. So I compare 40 with 39 it is a null pointer which means I have to put it right here. There is space so I put it there.

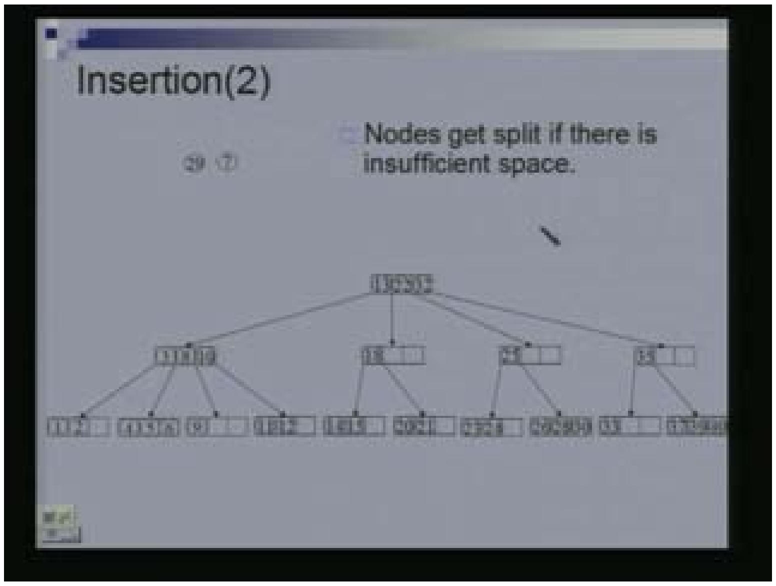
(Refer Slide Time: 23:48)



Insertion

- No problem if the node has empty space

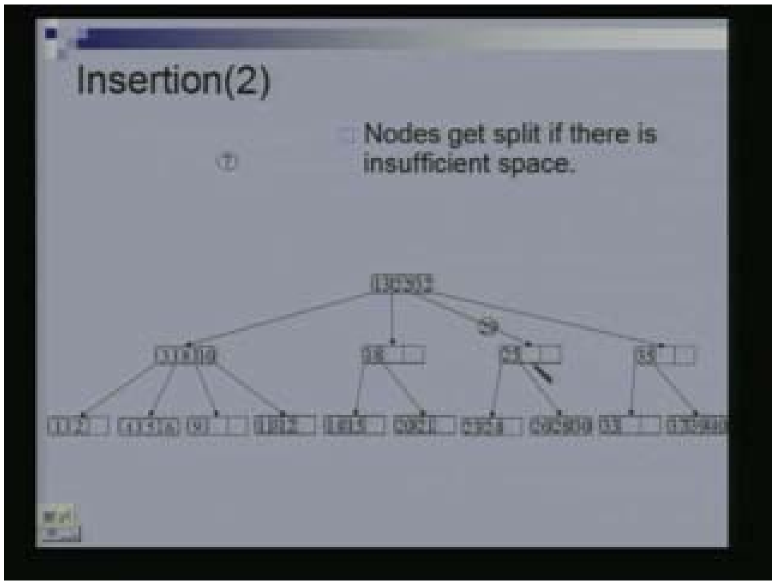
(Refer Slide Time: 23:58)


Insertion(2)

- Nodes get split if there is insufficient space.

If I am trying to insert a key and there is no space available in the node in which the key should go. Then what do I do and that is an example. When I am inserting 29 that is the kind of thing would happen. So 29 between 22 and 32 so I follow this. 29 more than 25 so it wants to come and sit here between 28 and 30 except there is no space here.

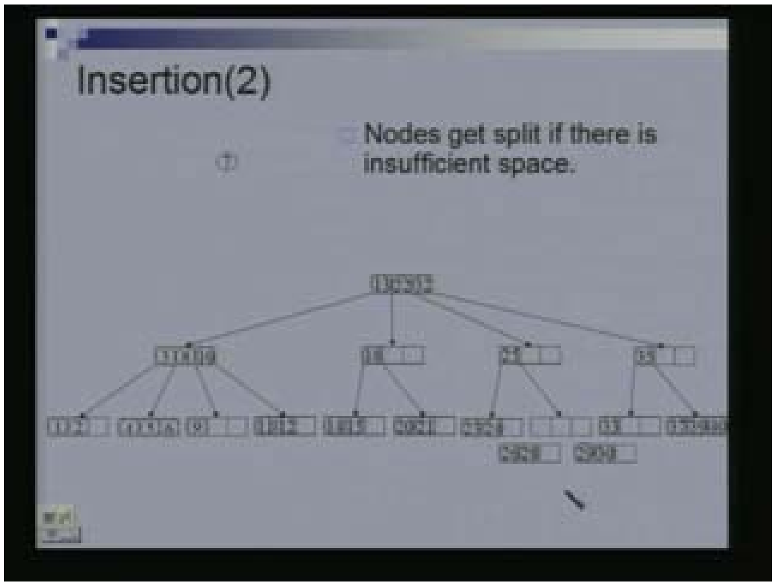
(Refer Slide Time: 24:20)


Insertion(2)

- Nodes get split if there is insufficient space.

So this is what we are going to do. We are going to split the node. Which node are we going to split? The one containing 26, 28, 29 and 30. We are going to split it in to 2. Let’s say these are 4 keys, the two smaller one will go to the left and two larger one will go to the right and we will remove this node. We need to link up this node, this should be the children of this guy here.

(Refer Slide Time: 24:53)

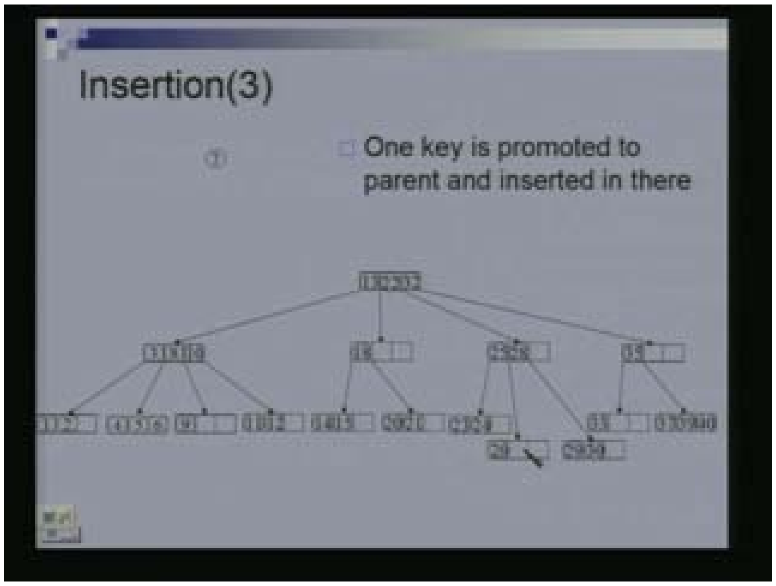


Insertion(2)

- Nodes get split if there is insufficient space.

Because these are all originally children of this node. So this should also be a child of this node but now its going to have 3 children. But how many keys are there? One, so we need one more key. If it has 3 children, it should have 2 keys. So which key I should put here? I am going to promote (Hindi conversation). So it is best to just promote up 28. That is what I will promote 28 here.

(Refer Slide Time: 25:54)

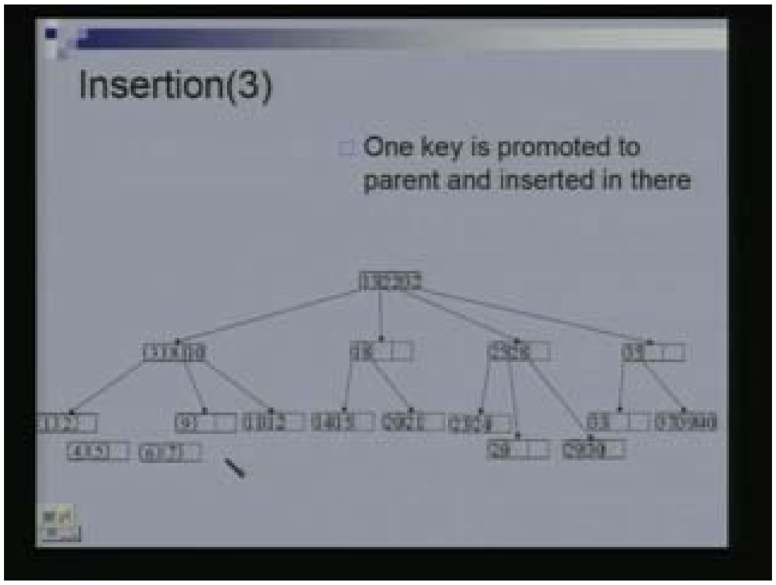


Insertion(3)

- One key is promoted to parent and inserted in there

I could also have promoted up 29. You understand why 28 and why not 26. If I had promoted up 26 what would be the problem? Then the search property would not be valid. So I have to promote either the largest key from this node, up here or smallest from here. This will become the new structure. We have promoted one key to the parent and inserted that key. We could insert the key in to the parent because there was a space in the parent. But it might happen that when I am trying to insert the key in to the parent, the parent does not have any space. (Hindi conversation) 7 less than 13 so we have go to left. The 7 between 3 and 8 so we should follow the second pointer. It should come here and we want to put it here except that there is no space, so we will split this node. Two nodes created. 4, 5 go to the left node 6, 7 to the right node. We get rid off this. These are the 5 children of this node (Hindi conversation).

(Refer Slide Time: 27:20)



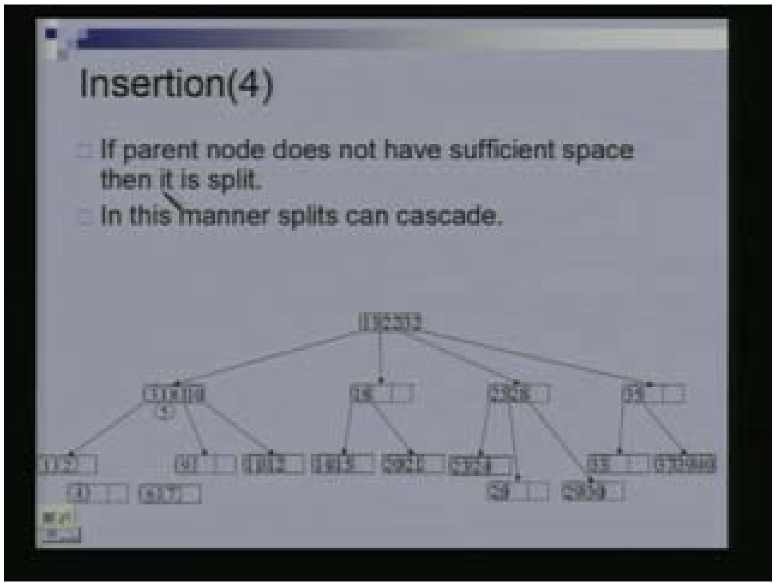
Insertion(3)

- One key is promoted to parent and inserted in there

So if the parent node does not have sufficient space then it is split. So we split the parent node in to two. 3 and 5 will go to the smaller one, 8 and 10 will go to the larger one that is to the other node. I have 1, 2, 3, 4, 5 children and they have to be made children of these guys. And one of the smaller (Hindi conversation) that has to be promoted up because when a split happens then we take the largest key of the smaller node and promote it up. The first two children would be made the children of this node. The right three would be made the children of this node. Why two of this and three of this? Because 5 is going to be promoted up.

So that means there is only one key left here which means that this can have only 2 children. The first two children will go here, 5 is going to be promoted up so this will have two keys which means 3 children. So these 3 would be its children and we promote 5 up. So we split this node (Hindi conversation) that we split first then we went and split the parent and now we will see the split happening here also. Because this does not have any space. So we will split it in to 2. The 5 and 13 would go in to one node. 22 and 32 will go to the other node. This will disappear (Refer Slide Time: 29:41) and now 1, 2, 3, 4, 5 these are five children.

(Refer Slide Time: 27:35)

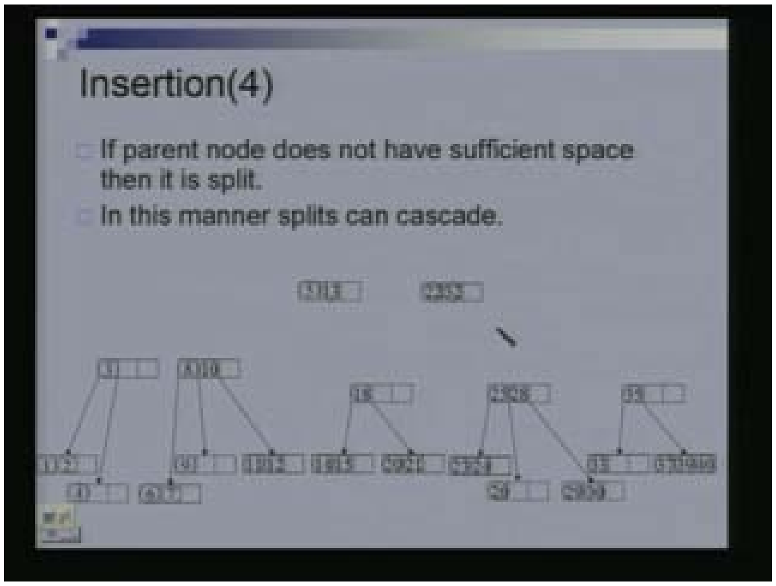


Insertion(4)

- If parent node does not have sufficient space then it is split.
- In this manner splits can cascade.

So 13 will get promoted up now. So the first two will become children of this and the next three would become children of this and 13 get promoted. But where does it get promoted? There is nothing above. So we will create a new root, eventually we may have to create a new root. That is what going to happen now. We create a new root, 13 goes up there and these two become the children of this.

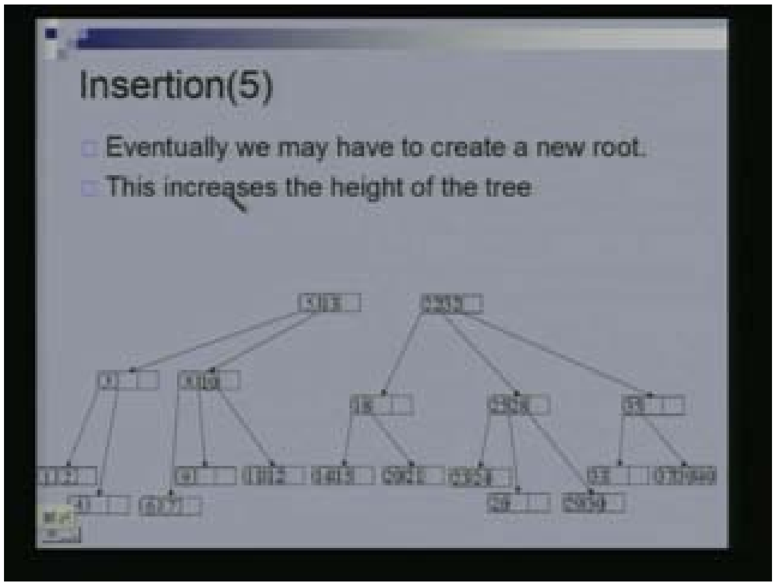
(Refer Slide Time: 29:40)



Insertion(4)

- If parent node does not have sufficient space then it is split.
- In this manner splits can cascade.

(Refer Slide Time: 30:05)

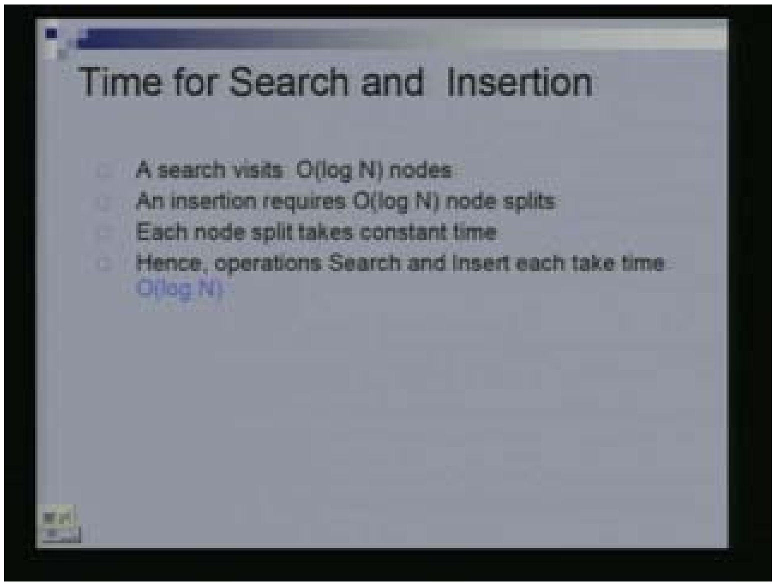


Insertion(5)

- Eventually we may have to create a new root.
- This increases the height of the tree

So if we create the new root the height of the tree increases by one. You understand the procedure so you first come down the tree till you hit a leaf. You will try to put the key there, if there is space nothing to be done. It is very simple. If there is no space then you split that node and then we decided that the two lower keys will go to one node and two higher keys will go to the other node. The largest key in the lower part would be promoted up. So when we split there are four keys in a node. There were four keys (Hindi conversation) to which means the second key of those four is the one which will get promoted up. Promoted up means we are trying to insert the key in to the parent node. If we are successful if there is a space no problem, otherwise repeat the split process of the parent node. So split it and this split might cascade all the way up to the root. If it cascades up to the root and the root also gets split then we have to create the new root. That is it.

(Refer Slide Time: 32:33)

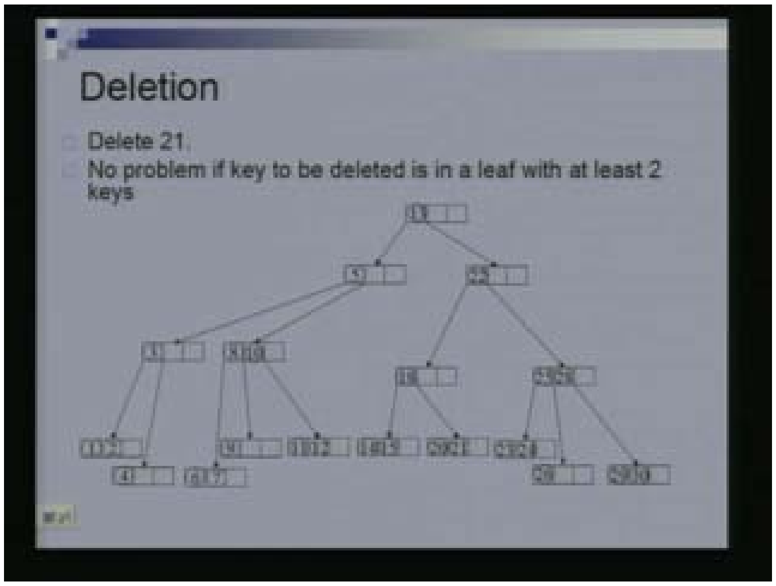


Time for Search and Insertion

- A search visits O(log N) nodes
- An insertion requires O(log N) node splits
- Each node split takes constant time
- Hence, operations Search and Insert each take time O(log N)

How much time does insertion take? So search was very clear. For search we said it will take order log n time. How many splits can be there in the process of insertion? Its at each level we might be doing the split. How much time does one split take? How much time does it take for me to split a node? I will create some two nodes (Hindi conversation) constant time independent of the number of node. So each node split takes constant time. So the total time is order log n (Hindi conversation).

(Refer Slide Time: 33:45)



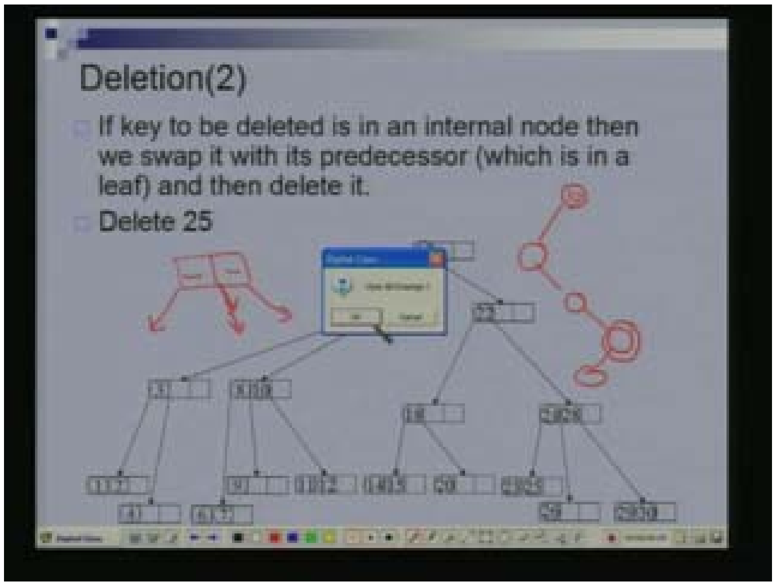
Deletion

- Delete 21
- No problem if key to be deleted is in a leaf with at least 2 keys

So now let's look at deletion. So suppose I wanted to delete 21. First as in the case of the binary search tree, first you have to search for 21. Find out where the key is. In the case of a binary search tree, recall deletion require 3 different cases. If the key was at the leaf then we just knock out the leaf, nothing to be done. If it was at the internal node then you had to distinguish between one child and two child. The one child case is not really happening here. So it is only two child case that we have to be worried about. If it is such an internal node with two children then what did we do? We found the successor or predecessor of that key lets say we found the predecessor and we move the predecessor to that and delete the predecessor. That is what we would mean. We are going to do something similar here.

Let us see. Suppose I wanted to delete 21. So 21 there is no problem because 21 is in a node. We will search for 21. We come down here, go right, go left, go right and I find 21 here. Why is there no problem in deleting 21? It is in a leaf node, I can just remove it and I can remove it without violating the property of the 2-4 tree. In a 2-4 tree we require each node has at least one key and at most three keys. So after deleting, this will still continue to have one key, so no problem. That is what is going to happen. I have not shown the process but this 21 will get deleted, we just knock it out from here, nothing to be done. If the key to be deleted is an internal node, is at an internal node. For instance I was trying to delete 25. I search for 25, I find 25 right here. What do I do? I am going to swap it with its predecessor. What is the predecessor of 25? How do I find the predecessor of 25 in the case of a 2-4 tree? I will go left and then keep going right (Hindi conversation) then I find the largest key in this node.

(Refer Slide Time: 37:35)



Deletion(2)

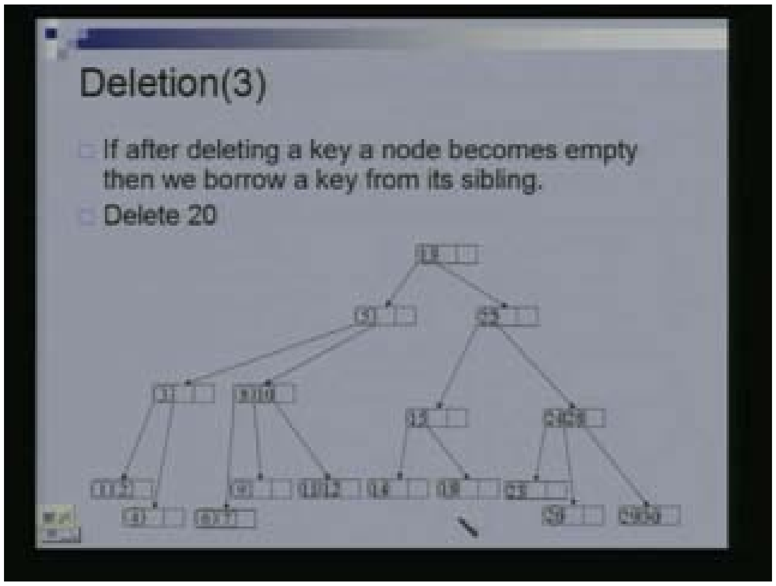
- If key to be deleted is in an internal node then we swap it with its predecessor (which is in a leaf) and then delete it.
- Delete 25

I find the largest key its 24, so predecessor of 25 has to be 24. I am going to swap this two. Then I am going to remove the 25 from here. This is a same thing, I can remove the 25 from here. Why, because its a leaf already. There are two keys in the leaf, so if I remove one there is no problem. Note that the predecessor will always be in the leaf in this case. Not in the case of a binary search tree. Let us check this point out. In the case of the binary search tree, the predecessor of a node need not be a leaf node. Suppose this was my binary search tree, this was a node 10. I am finding the predecessor so I go left and then I go right. This could have been my binary search tree. What would be the predecessor of 10? It would be this guy (Refer Slide Time: 36:15). This is not a leaf node but here the predecessor will always be a leaf node. Why? How do I find the predecessor? I go left and then keep going right, keep taking the right most child. So when will I stop? When there is no right child. What does it mean when there is no right child? When my right child is null then that means all the children are null. Because I cannot have a situation in which there is a node which has a key, it has no right child but it has left child. This is not permitted at all.

If there is a key then we will have two children, if there were two keys then we will have three children and so on. So my predecessor would always been a leaf, so I would just remove that leaf node and I would be done. So that is what I do. Recall I was deleting 25, so I swapped 24 and 25 and now I have to just get rid off 25. So I will get rid off 25 by that. So 25 disappeared from here. It is a very simple case. As you can imagine, problem were arising when I am in a leaf, I am trying to delete a key from a leaf which has only that one key in it. Then that leaf becomes empty so what do I do now? So let us look at that.

If after deleting a key, a node becomes empty then we borrow a key from its sibling. Let us see what that means. Suppose I am trying to delete 20. So I search for 20, I come down in this manner. I reach here, now if I delete 20. So 20 is removed, problem is this is an empty node not permitted. What will I do? Borrow from sibling. What does borrow from sibling mean? This guy has only one sibling. So can I borrow 15 from here to here? No, it is disaster because search property is not going to be valid. So we are going to do something like a rotation like we did in an AVL tree. What does it mean? 15 goes up and 18 comes down.

(Refer Slide Time: 38:23)

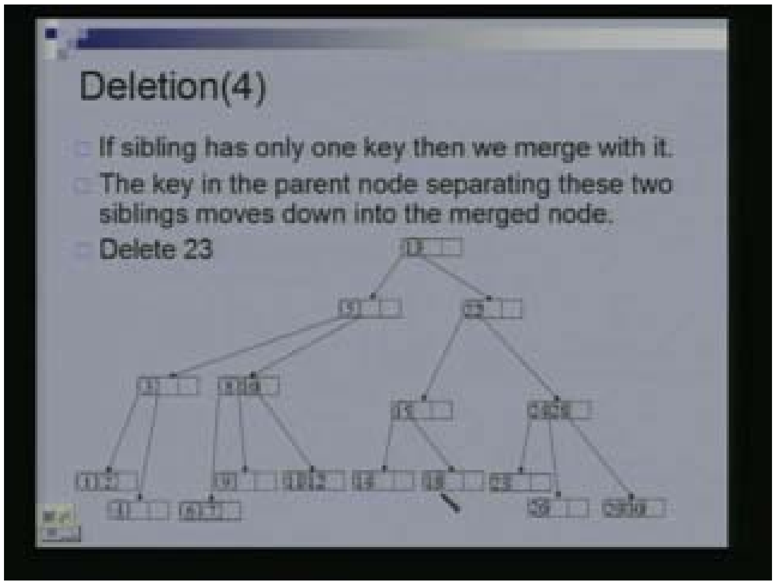


Deletion(3)

- If after deleting a key a node becomes empty then we borrow a key from its sibling.
- Delete 20

Then the next thing you are wondering is, if I can not borrow from my sibling. When can I not borrow from my sibling? When the sibling has only one key in it. For instance now if I were trying to delete 18 then that would be a problem. Let see. If sibling has only one key then we merge with the sibling that is we combine with the sibling. Suppose I was trying to delete 23. So 23 is here if I delete it this is an empty node. So I try to borrow from a sibling.

(Refer Slide Time: 39: 44)



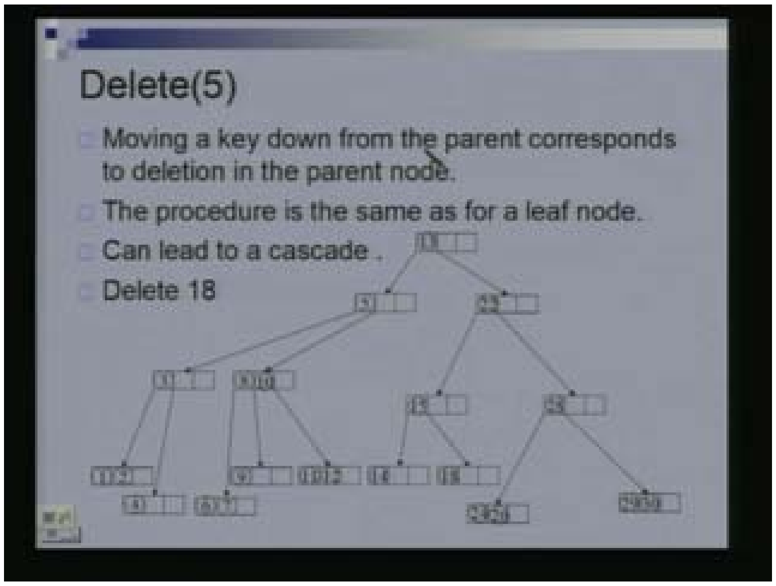
Deletion(4)

- If sibling has only one key then we merge with it.
- The key in the parent node separating these two siblings moves down into the merged node.
- Delete 23

There is a small catch here (Hindi conversation) one and two, but I cannot borrow from this one. You see why it is. Because if I have to promote something then 28 is going to come down but this going to jumble this. So when I say borrow from a sibling I really mean and adjacent sibling. (Refer Slide Time: 40:35) I can borrow from here, so when I am here I can borrow from here, if I am here I can borrow from here. If I am here I can borrow from here or from here. But here I can only borrow from this guy. Did everyone understand why this is required? I try to borrow from here but if I borrow from here that is 26 goes up and 24 come down this is going to become empty. So that is not going to solve our problem. What we are going to do is merge. These two are going to merge, combine but if these two combine then the number of children here of this guy will become two which means (Hindi conversation). So the key in the parent node which separates these two siblings is this key, which is separating these two siblings is going to move down in to the merge node.

Let us see, I create a new node which is the merge of these two nodes. This 24 moves down and this 26 also. These are the only keys in the new node. Because there was one here, there was none here and there was none here. Two keys in all so they come and sit here. These two will now disappear and this becomes the child of this node (Hindi conversation). Now what can happen? Essentially what we have done is we said we are going to one of the keys from the parent node is going to come down. But what if there was only one key in the parent node. The same as before (Hindi conversation) and so on. So moving the key down from the parent node corresponds to deleting the key from the parent node.

(Refer Slide Time: 43:02)



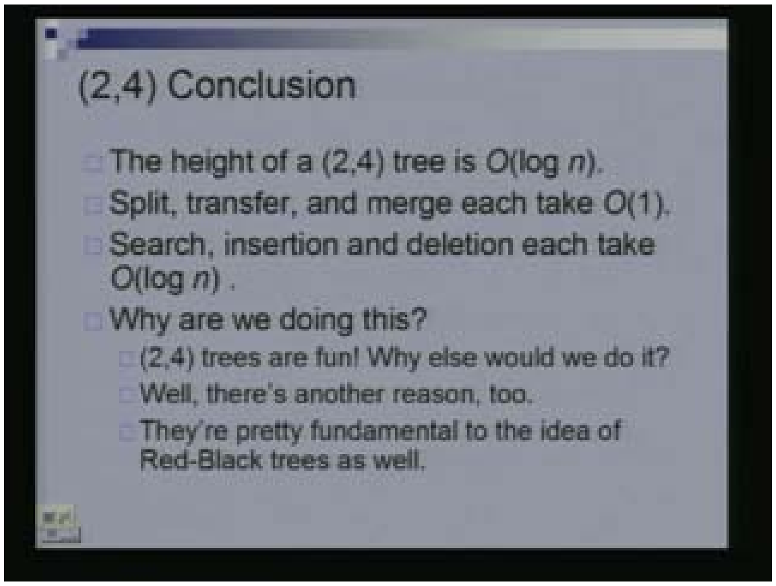
Delete(5)

- Moving a key down from the parent corresponds to deletion in the parent node.
- The procedure is the same as for a leaf node.
- Can lead to a cascade .
- Delete 18

This procedure will be the same as that we have done so far in this leaf node. But it can lead to the cascading. Cascading as in (Hindi conversation) we will see that happening in this example (Hindi conversation). This is the only child left of its parent (Hindi conversation) sibling is also only one key. So we are going to merge with the sibling. We are going to create a new node which is going to get the sibling key and the key from the parent which is 22. This is the situation now and we will delete this and this (Hindi conversation).

Because of these deletions, height can reduce by one. After all we said height (Hindi conversation) $\log_4 n$ is less than what it was, then that means height has to shrink and that would happen. There are just very few concepts that we really handled and insertion (Hindi conversation) or deletion (Hindi Conversation). (Hindi conversation) no point is doing that. So first you try to borrow if not successful then you merge. So let us conclude today's discussion. Height of a 2-4 tree we have seen is log n.

(Refer Slide Time: 48:15)



(2,4) Conclusion

- The height of a (2,4) tree is $O(\log n)$.
- Split, transfer, and merge each take $O(1)$.
- Search, insertion and deletion each take $O(\log n)$.
- Why are we doing this?
  - (2,4) trees are fun! Why else would we do it?
  - Well, there's another reason, too.
  - They're pretty fundamental to the idea of Red-Black trees as well.

This would actually be a theta of log n because it is at least $\log_4 n$ and at most $\log_2 n$. So as far as deletion was concerned we have not looked at running time for deletion yet. But you can see that is also log n. Why? It was first we come down the tree to search for the key that is log n and then we keep moving back up. At every step we might go all the way back up to the root. So another log n step. Each step where either borrowing one from the sibling or we are merging with the sibling. But all of them are constant time operations. Borrowing would correspond to (Hindi conversation) constant time (Hindi conversation). So what you have seen is that search, insertion and deletion all take order log n time in a 2-4 tree. So why did we come up with this very complicated data structure? Why are we doing all of this? There is another reason, we are going to see another data structure called red black trees. And that is also very fast data structure for implementing dictionaries. What we are going to see is that what we learnt about 2-4 trees today is going to be very helpful in understanding how the red black trees functions. So we are going to look at this in next class. So with we will end today’s discussion on 2-4 trees.