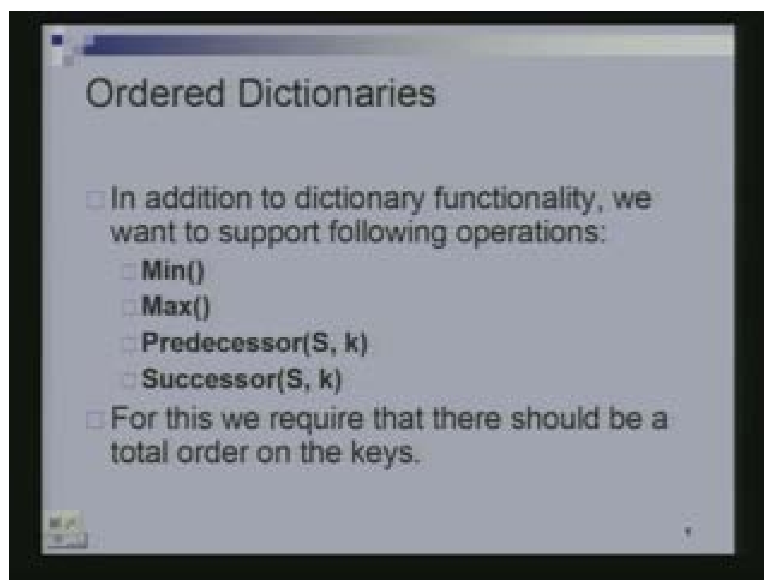
Data Structures and Algorithms
Dr. Naveen Garg
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi

Lecture – 8
Ordered Dictionaries

In today's class we will be talking about ordered dictionaries. We will also be looking at binary search tree which is perhaps one of the simplest ways of implementing an ordered dictionary.

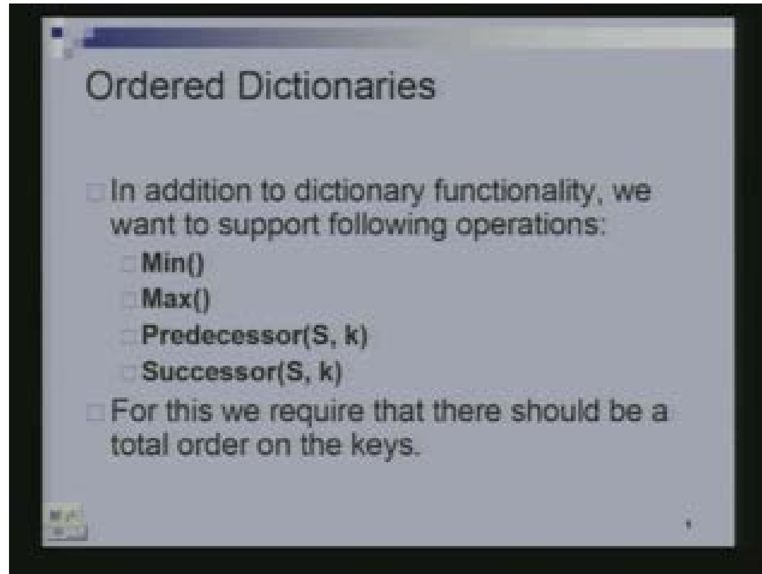
(Refer Slide Time 01:16)



So what is an ordered dictionary? It is essentially that you have the dictionary functionality. Recall dictionary was that you have key element pairs and you use the to insert an element, to search for an element or to delete an element. Besides that, in an ordered dictionary, you have the notion of the element with the minimum key, the element with the maximum key and the notion of predecessor and a successor. So when I say minimum key what then it means that there is some kind of a total order on the keys. This is different from when we have talked about hashing and dictionary. If we recall we said that the only operation that will ever need to do on the keys is to compare two keys for equality. Given two keys, we have to decide whether they are the same or not. So here we are going to be excepting more out of our keys.

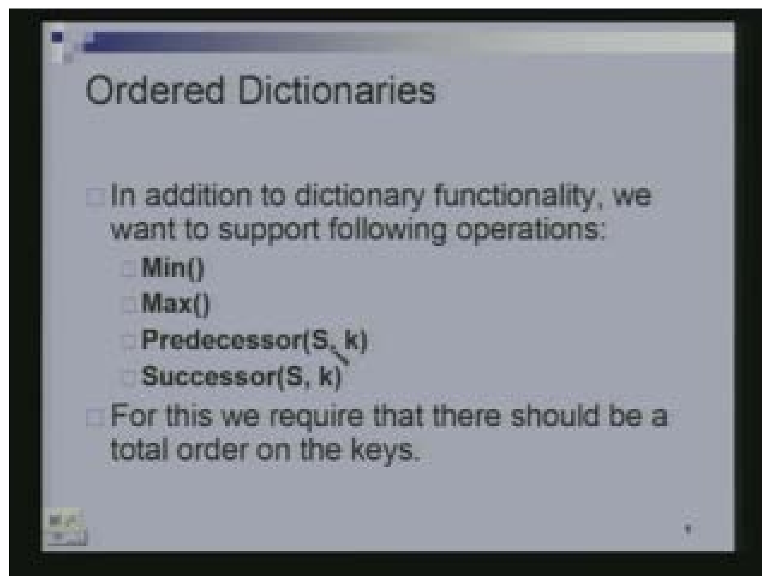
Here we are going to assume that there is some kind of a ordering relation on the keys so that given two keys, I can either decide whether they are equal and if they are not equal then I can say one is less than the other one is larger than the other. There is ordering relation on the keys.

(Refer Slide Time 01:28)



Conversation between a student and professor: what are the parameters? You're saying to the function predecessor and successor. I have just shown 'S' as the ordered dictionary as one of the parameters it need not be. So in particular you could think of a predecessor as taking only one parameter which is a particular key. So predecessor of k would give me the key which precedes k in the total order in the dictionary.

(Refer Slide Time 02:50)



There are certain keys in dictionary s. Give me the one which precedes the key k and which lies in this dictionary. Keys are the particular field or the part of data on which the things are ordered. There is a total ordered on the keys. Similarly the successor function.

(Refer Slide Time 03:34)

Ordered Dictionaries

- In addition to dictionary functionality, we want to support following operations:
  - **Min()**
  - **Max()**
  - **Predecessor(S, k)**
  - **Successor(S, k)**
- For this we require that there should be a total order on the keys.

So what is one way you can implement such an ordered dictionary? There are two trivial ways of doing it where in both cases, we say using a list kind of data structure. So an unordered list would just keep all the elements in here. So all I have shown here are the keys. Since it is unordered, inserting takes only a constant amount of time. Searching will take ordered 'n' time in the worst case because there is no order. I might have to go through the entire list before I found out where the element is and how much time will deletion take? It takes order 'n' because first we have to search for what we are trying to delete.

(Refer Slide Time 03:45)

A List-Based Implementation

- Unordered list
  - searching takes $O(n)$ time
  - inserting takes $O(1)$ time

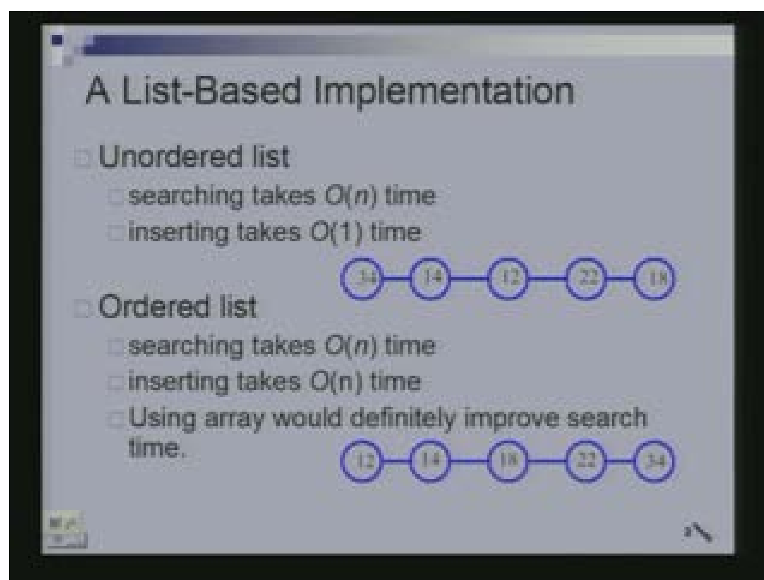
34 — 14 — 12 — 22 — 18

- Ordered list
  - searching takes $O(n)$ time
  - inserting takes O(n) time
  - Using array would definitely improve search time.

12 — 14 — 18 — 22 — 34

How much time would the successor take here? Order n. not order one because suppose I say what is the successor of 12 in this dictionary? Successor of 12 is not 22. It is 14 which is the key following 12 in the order relation. The order relation I am assuming is just the order on the integers. So 12 is the smallest key, 14 is the key larger than that. 18, 22, 34. So 34 is the largest key in it. Given the key 12, I have to run through this entire thing to find out the one which was the smallest key larger than 12. Both successor and predecessor and min and max will all take order n time. So it's fairly an efficient implementation. An ordered list on the other hand, let's say we ordered the thing according to the total order on the keys. So now minimum takes only constant time. Maximum also we can organize so that it takes constant time. Let's say I have a pointer reference to the end node in this list.

(Refer Slide Time 04:33)



Successor also takes constant time because given a particular node, I can just go to next one that will give me the successor. Let's say predecessor takes constant time. It depends upon how you give the node. When I ask for successor of a node if you have to search for the node then of course it will take order n time because you might have to go through this entire list to reach that node. But if I tell you where the node is, let's say I give you a reference to this particular node in this list, then you can compute the successor and predecessor in constant time. Inserting also takes order 'n' times because you have to find out where to insert. You have to find out the correct position for insertion. Searching takes order 'n' time because again we may have to run through the entire thing. If we use an array, searching can improve and you have seen an example of how to do this.

(Refer Slide Time 05:43)

A List-Based Implementation

- Unordered list
  - searching takes $O(n)$ time
  - inserting takes $O(1)$ time

34 — 14 — 12 — 22 — 18

- Ordered list
  - searching takes $O(n)$ time
  - inserting takes O(n) time
  - Using array would definitely improve search time.

12 — 14 — 18 — 22 — 34

(Refer Slide Time 06:03)

A List-Based Implementation

- Unordered list
  - searching takes $O(n)$ time
  - inserting takes $O(1)$ time

34 — 14 — 12 — 22 — 18

- Ordered list
  - searching takes $O(n)$ time
  - inserting takes O(n) time
  - Using array would definitely improve search time.

12 — 14 — 18 — 22 — 34

We use binary search to do that. If you put the elements they are ordered. I put them in an array. Then we can do a binary search to find the element in log time. If we do binary search then now insertion and deletion still take order n time.

(Refer Slide Time 06:33)

Binary Search

- Narrow down the search range in stages
  - findElement(22)

While we can find out what is the place to insert an element, you will have to shift all the elements to the right. So it takes lot of time to do the insertion. Similarly for deletion, we know where the element is which we are trying to delete. Then we have to shift everything to the left. Just to recap what binary search was, so you remember binary search. To search for 22, you go to the middle element. 22 is larger you will go to the right. 25 which is smaller go to the left so on and on. In that manner, you will eventually end up with the array location which has 22 or an array location which does not have 22. In that case able to say whether 22 is there or not and the number of comparisons you would take in this process is only logarithmic.

(Refer Slide Time 07:15)

Binary Search

- Narrow down the search range in stages
  - findElement(22)

(Refer Slide Time 07:43)

Binary Search

- Narrow down the search range in stages
  - findElement(22)

Every time you make a comparison the size of the array and which you are making the search 'halves' and number of times you can half n to get down to one has only log of n and so you will take order log n time to do the searching. Recall insertion and deletion I said that will take order n time. These are the trivial ways you adopt to implement an ordered dictionary.

(Refer Slide Time 07:49)

Running Time

- The range of candidate items to be searched is halved after comparing the key with the middle element
- Binary search runs in $O(\lg n)$ time (remember recurrence...)
- What about insertion and deletion?

But we are not here to talk about trivialities. So we are going to look at something more interesting and that's called a binary search tree. What is a binary search tree? A binary search tree is a binary tree which has a search property on it. Recall what is the binary tree. Binary tree is a tree in which every node has at most two children. A node can have one child, two children

or no children. No children means it is a leaf node and now there is a search property that we are talking out.
So each node is going to contain a key and an element. In most discussions that follow, we will not be talking about the element at all. We are just interested in the keys.

(Refer Slide Time 08:32)

Binary Search Trees

- A binary search tree is a binary tree T such that
  - each internal node stores an item (*k*,*e*) of a dictionary
  - keys stored at nodes in the **left subtree** of *v* are **less than or equal** to *k*
  - keys stored at nodes in the **right subtree** of *v* **are greater than or equal** to *k*
- Example sequence 2,3,5,5,7,8

What is written on the nodes are the keys. Now what is a binary search property? The binary search property says the following. All the keys which are less than five will be in the left sub tree. All keys which are larger than five will be in the right sub tree and this property holds at every node. Not just at the root node. [HINDI] not all keys which are larger than 3. Of course there are keys larger than 3 which are here but basically if I look at the left sub tree, all the keys in the left sub tree will be less than this node. All the keys in the right sub tree will be larger than this node (Refer Slide Time: 09:47). Similarly for this tree. All the keys in the right sub tree as you can see are larger than this key value. If I look at this node, all keys in the left sub tree are less than 7. All keys in the right sub tree are larger than 7. This is called search property.

(Refer Slide Time 09:05)

A binary tree in which the nodes have keys which satisfy the search property. So the search property satisfied is called a binary search tree. So binary tree plus search property equals binary search tree. I have these set of keys: 2, 3, 5,5,7,8. This is the binary search tree with these keys in it. This is also another binary search tree with the same set of keys.
You can have many different kinds of trees with the same set of keys. Both of them are binary search trees because they both are binary and satisfy the search property.

(Refer Slide Time 10:26)

Conversation between Professor and student: It is implementation. Keys stored at the node in the left sub tree of v are less than or equal to k. (Refer Slide Time: 11:00) So there are couple of

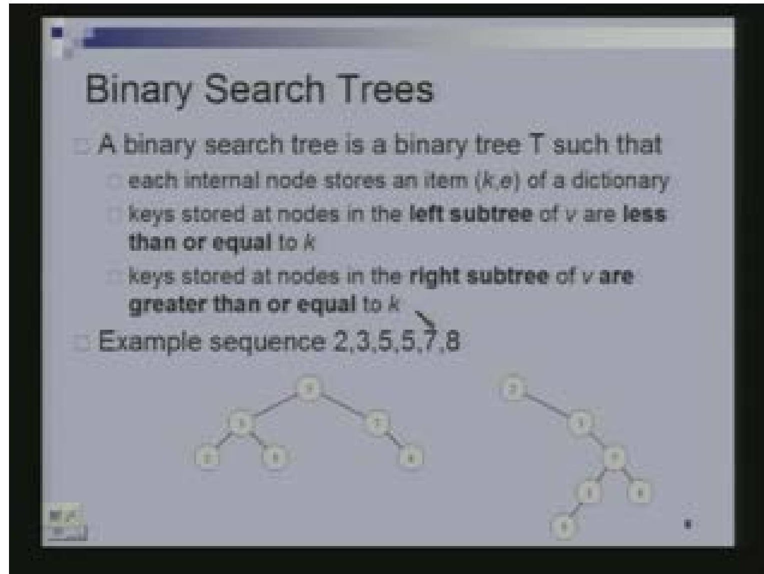
features here. Here I am assuming that you might have two keys which are the same. So quite often this doesn't happen. So quite often you would be using dictionaries only in settings were the keys are unique. No two keys are the same.
(Refer Slide Time 11:22)

Binary Search Trees

- A binary search tree is a binary tree T such that
  - each internal node stores an item (*k*,*e*) of a dictionary
  - keys stored at nodes in the **left subtree** of *v* are **less than or equal** to *k*
  - keys stored at nodes in the **right subtree** of *v* **are greater than or equal** to *k*
- Example sequence 2,3,5,5,7,8

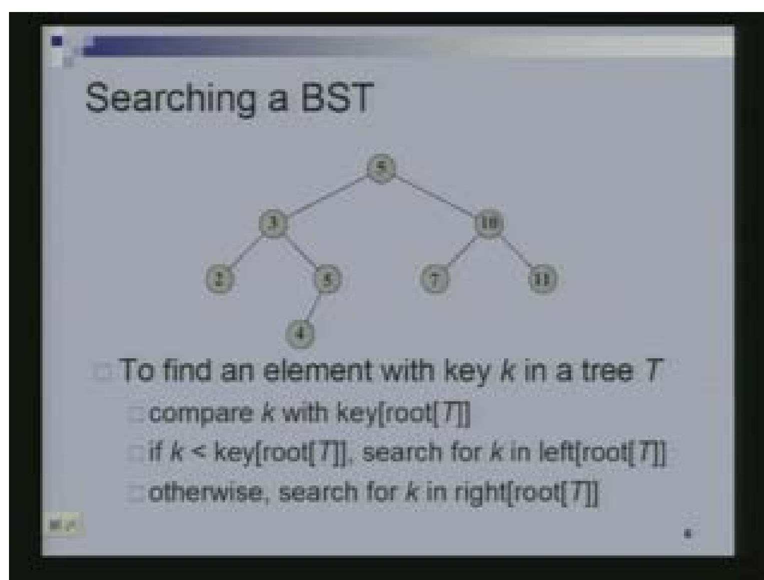
Suppose you had the setting where two keys could be the same, suppose your key was the name of a student, you can define total order of names let's say lexico graphic order or alphabetic order and then two names could be the same. So you could have settings in which the two keys are the same. Then we have to decide whether if a key is equal. Whether it should go to the left sub tree or it should go to the right sub tree and we can decide one way or other. Let's say we go to the left. We could easily have said it goes to the right. There is no problem with that. Here actually I am permitting it to go both ways. Either go to the left or it could go to the right.

(Refer Slide Time 11:54)



So for the rest of the discussion let me just assume the keys are unique. Otherwise it unnecessarily complicates matters. So we will just assume keys are unique, do the entire discussion see if there is a need of duplicate key. This is what should we do then. If there are duplicate keys how do you handle that now? Suppose you want to search in a binary search tree, given a particular key. So after all it's a dictionary. We are implementing dictionary. So given a particular key I want to find out where the element is. So let me first show you an example.

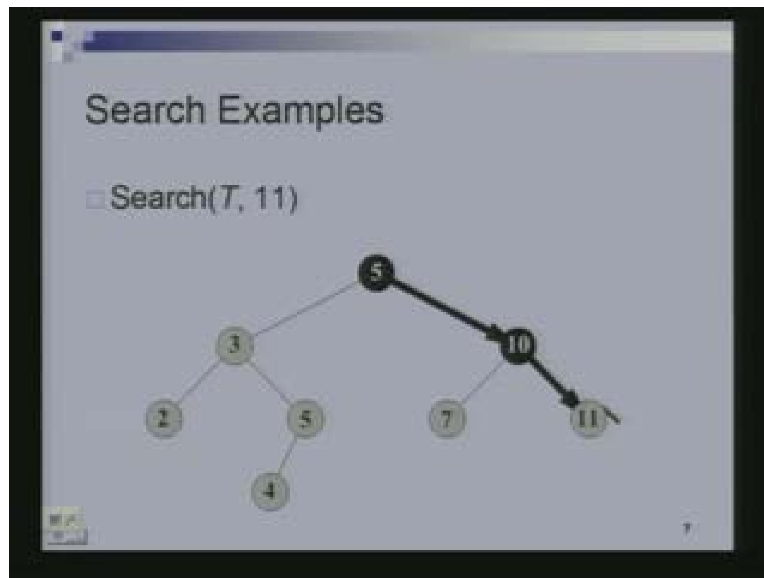
(Refer Slide Time 12:50)



Suppose this is my tree and i want to search for 11. I come to the root. I compare 11 with 5. 11 is larger than 5. So search property says that 11 has to be in the right sub tree. That was the

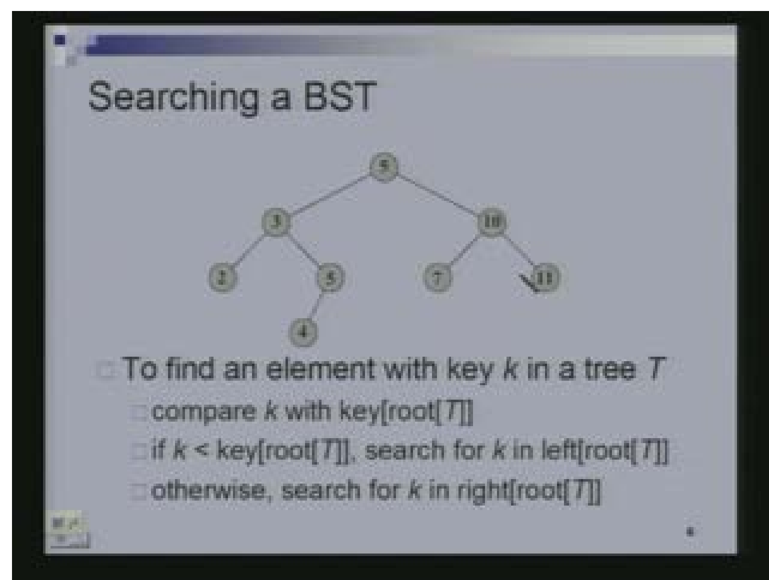
search property. Keys which are larger than here will be in the right. Keys which are smaller will be in the left. So we go to the right sub tree. [HINDI] 11 is larger than 10. So once again if it is there at all in the tree, it has to be in the right sub tree. Hence we go the right sub tree and we compare 11 with this and we find 11. So we are done. We found 11.

(Refer Slide Time 13:00)



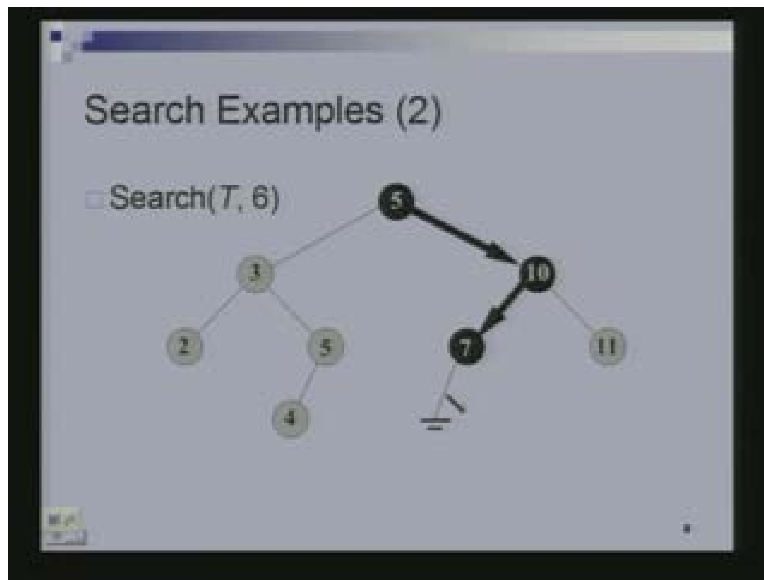
That's what was being said here. To find an element with a key k in a tree T, you will compare key k with the key in the root. If k is less than the key in the root then you will search for the key in the left sub tree. Otherwise you will search for k in the right sub tree.

(Refer Slide Time 13:54)



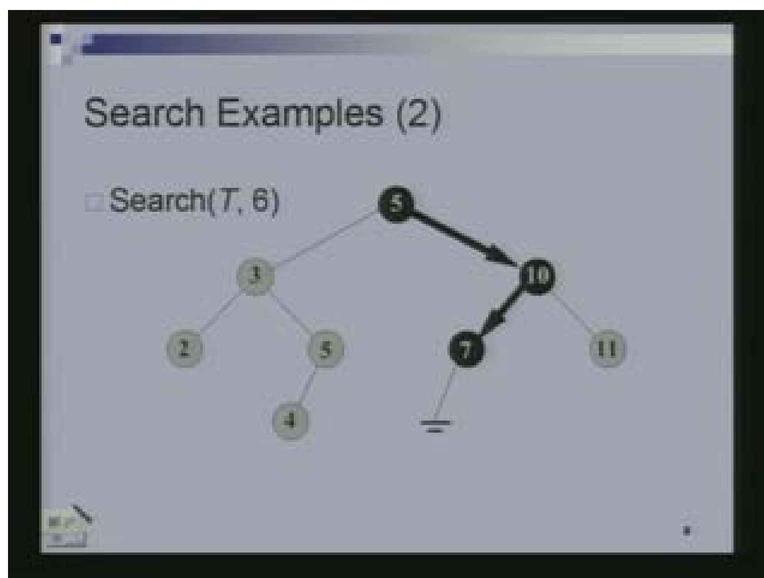
Suppose we have same example but now searching for 6. We come here 6 is larger so we go right. Then we compare 6 with 10. 6 is smaller. So we go left. Then we compare 6 with 7. 6 is smaller. So we try to go left. But the left child is null. So it's not there. Because if it were there, it has to be here.

(Refer Slide Time 14:22)



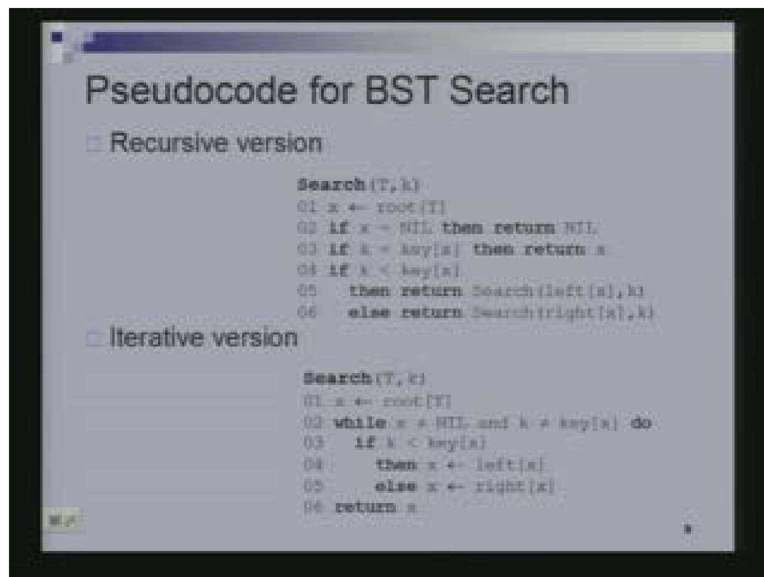
It has to be in the right sub tree of this guy. Because 6 is larger than 5. It has to be in the left sub tree of this guy because it's less than 10. It has to be in the left sub tree of 7 since it is less than 7. So if 6 were there, it had to be here and since it is not here, it's not there.

(Refer Slide Time 14:56)



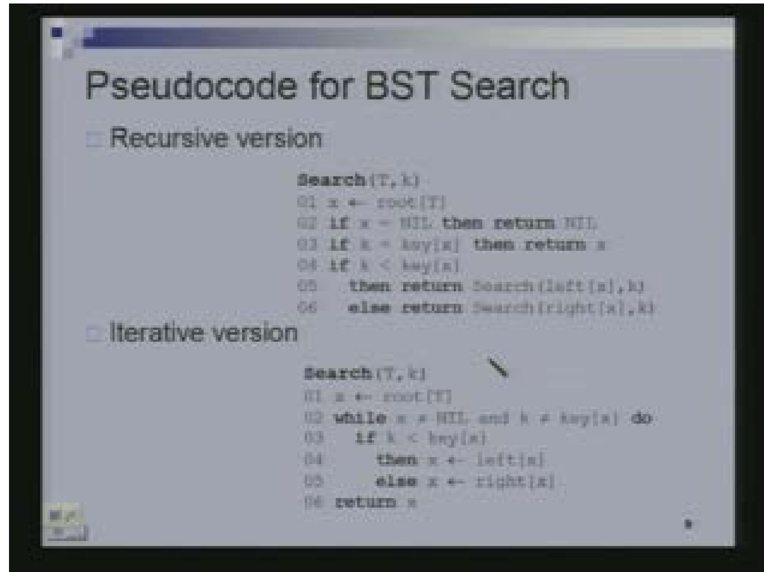
So we can write the search procedure for binary search tree either as a recursive procedure or as an iterative procedure. So the recursive procedure is perhaps the simplest to understand. So you are searching for a key k in a tree t. So you look at the root of T. Let's say that its x. If x equals nil which means there is no root or empty tree. Then you are essentially saying that nothing is there. So just forget this for now. If the key in this root node x is equal to the one you are searching for, then you just return the root node. If it is less than the key in the root node, if k is less than the key in the root node, then you have to recursively search in the left sub tree. So you are searching in the left sub tree of x. And what you are searching for? You are still searching for k.

(Refer Slide Time 16:20)



So what is the left of x doing? This is all pseudo code. Let me come to what your question is. X is a reference to a particular node. So x to begin with refers to the root of the tree and then left of x is the reference to the root of the left sub tree. So it's referring to that and we are searching for k in there. So that's what we have to do and if key is larger than the key in the root, then we have to search for k in the right sub tree. This is clear with everyone. Let's go to the iterative version. In the iterative version, we are not going to make recursive calls to the search procedure. What we are going to do is as we are doing in the search, we are just going keep matching down the same tree.

(Refer Slide Time 18:14)

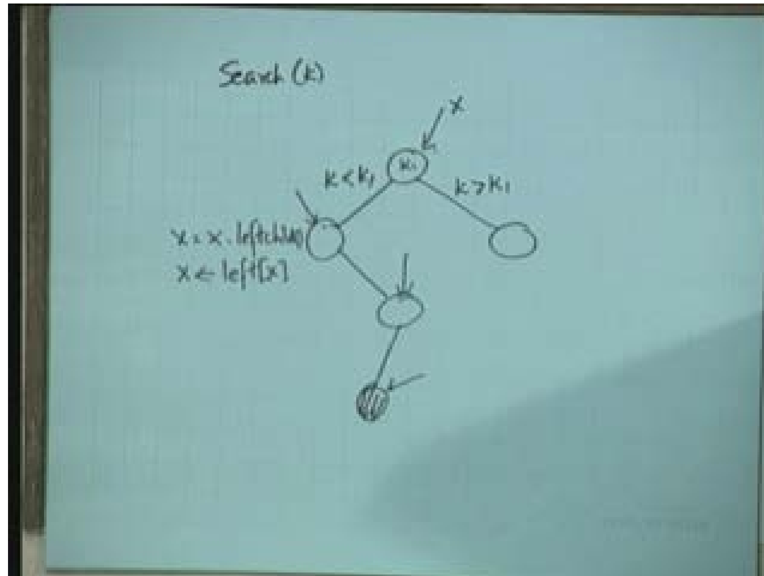


So we start with x referring to the root of the tree. Well if x is nil which means that it's an empty tree and k is not the key in the root. Then we will do something. What will we do if k is less than the key? Then we have to go left. So x now becomes the left child. x now gets the value of the left child of this current node.

So we started off with x, referring to this guy and then we said if k, so suppose there is key k_1 here we are searching for the key k. If k is less than k_1 then we go left, if k is more than k_1 we go right and since we go left here we are now continuing the search here. So x is gets the new value which is either x dot left child either you want to call this way or actually in the code just showed you just now, I have written it as x gets left x.

Pseudo code I can use any one of these so this is what it gets and we continue this search. We are basically may be the next step we go right and so on and on till we reach here, this is where our node x. So x will keep getting modified, first it will pointing to this node then its pointing to this node, pointing to this node and eventually pointing to this node. It's keep getting modified in this manner.

(Refer Slide Time 21:02)



How much time does search take? So let’s look at the iterative version of the search. What we did was that with each time we went through this while loop which is here, we came down one level in the tree. We went from a node to one of its child nodes either the left child of that node or the child of that node, [HINDI] we can came down one level.

(Refer Slide Time 21:15)

Pseudocode for BST Search

- Recursive version

```
Search(T,k)
01 x ← root[T]
02 if x = NIL then return NIL
03 if k = key[x] then return x
04 if k < key[x]
05    then return Search(left[x],k)
06    else return Search(right[x],k)
```

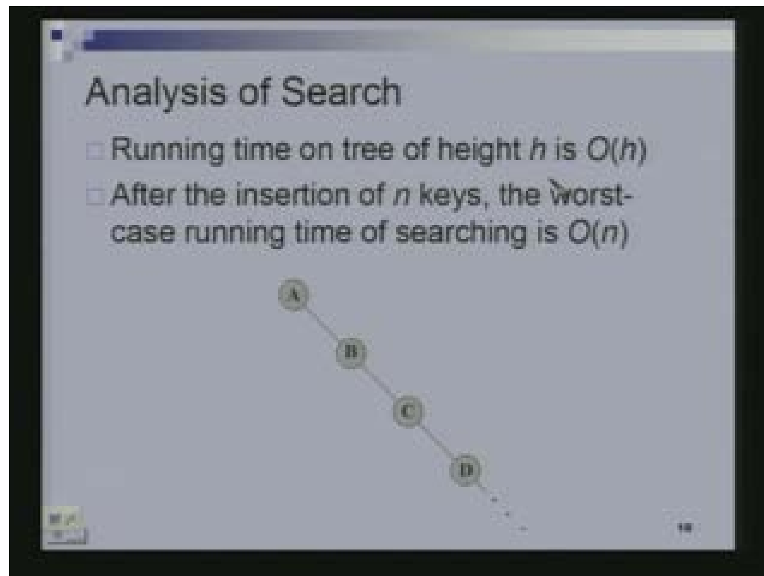
- Iterative version

```
Search(T,k)
01 x ← root[T]
02 while x ≠ NIL and k ≠ key[x] do
03    if k < key[x]
04       then x ← left[x]
05       else x ← right[x]
06 return x
```

We started at level zero which is the root then we came after one run of while loop, we came to one level after that we came to level 2, level 3 and so on. How many times we will execute this while loop, the maximum number of levels in the tree and what is a maximum number of levels

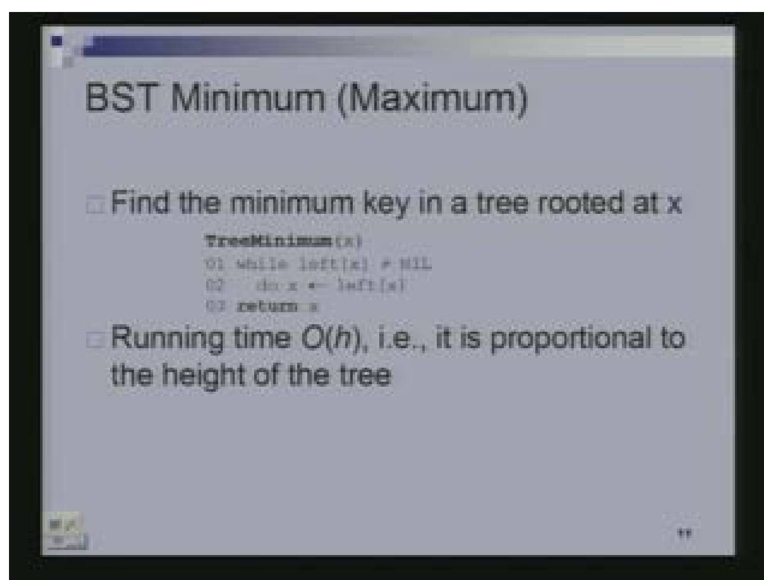
in a tree, it's the height of the tree. So if the height of the tree is h then the running time of procedure is no more than h. So order h is the running time of procedure.

(Refer Slide Time 21:50)



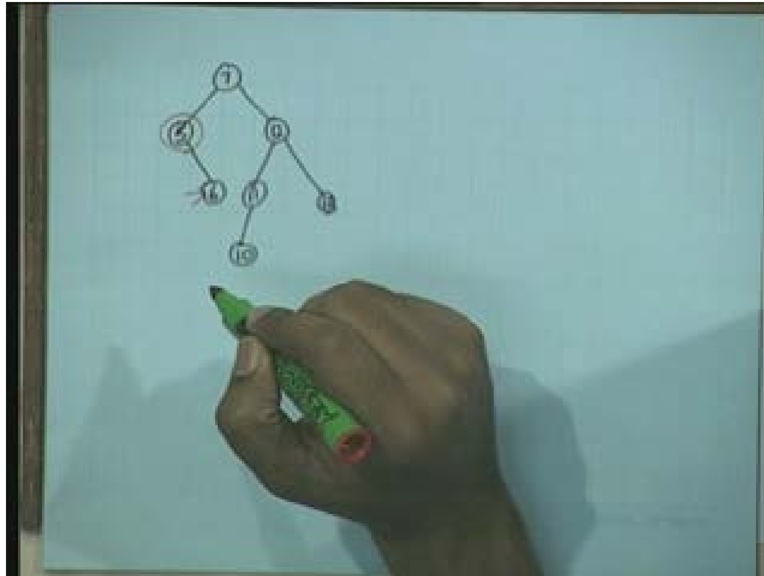
Now h recall can be very large. I might have a tree on n nodes whose height is order n and we will see at what kind of situations this happens but note that the height of the tree could be very large. The search time is only order h the height of the tree, order the height of the tree but the height of the tree in particular can be has large as the number of nodes in the tree.
Let's look at other procedure that of finding the minimum element in the tree.

(Refer Slide Time 22:45)



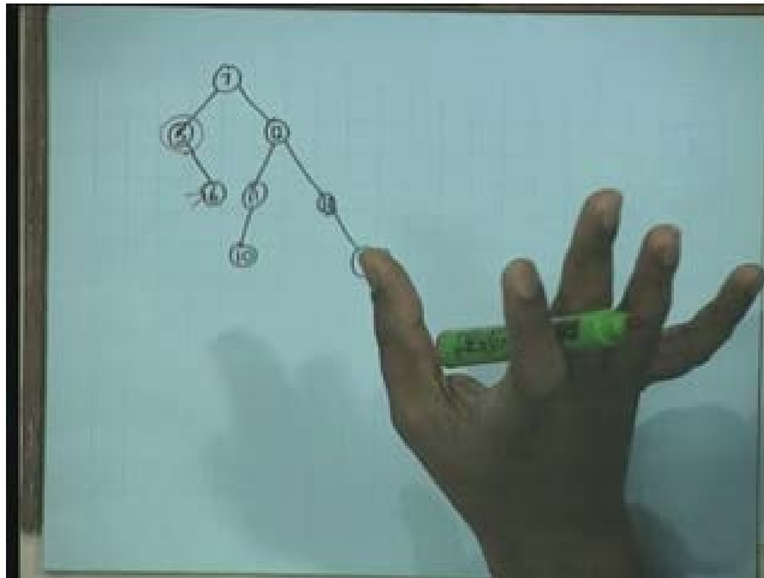
Where do you think the minimum element in a binary search tree? Why left most tree? Left most node or left most leaf? Left most leaf. Let’s see why that’s wrong. Suppose I have a tree which looks like the following. I need to put up in some keys so that this looks like a binary search tree. So let’s put in 7, 5, 6, 12, 13, 11, 10. This is the binary search tree? Now which is the smallest node here? The leftmost leaf, I do not know the left most leafs but this must be the left most leafs but this is not the smallest one. Smallest node is 5, it’s really the left most node expect left most node also doesn’t too much sense. Last leaf, this is not a leaf at all.

(Refer Slide Time 24:29)



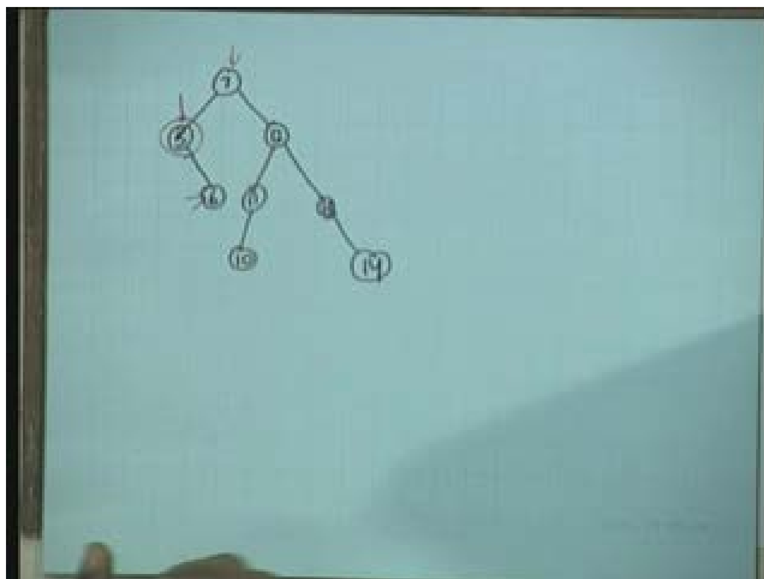
Let’s not try to say which node… last internal node. This is an internal node, you want me to take more internal node without left children. No, I can create more internal node without left children. So this is also an internal node without any left children so don’t try to just give the half a sentence definition of which the smallest is, but let’s give a procedural definition.

(Refer Slide Time 25:02)



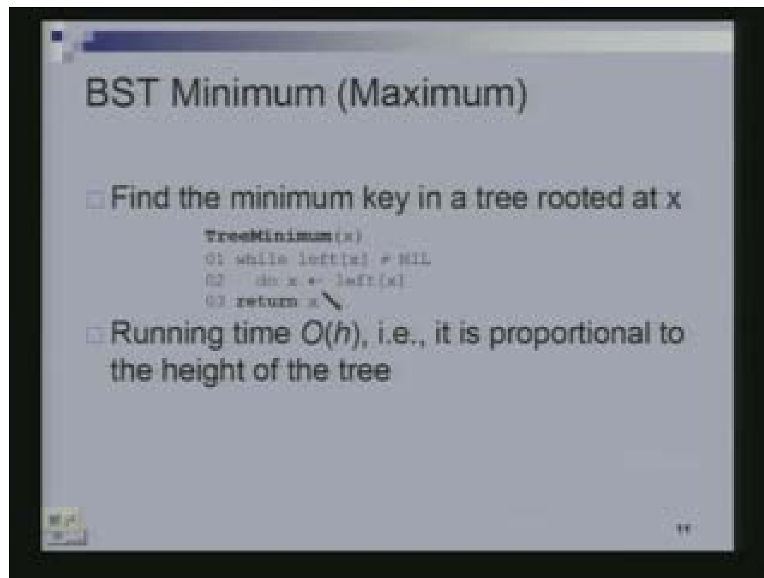
So which is how do you find the smallest? Start at the top, keep going left till left becomes null that is clear to everyone. So start from the top, keep going left till the left is null and that's the smallest node. How about maximum? Keep going right till the right becomes null. What's the proof? Why this is minimum? So the proof is very simple, the minimum has to be to the left of its root, so it has to be in the sub tree. Then you have smaller, so the smallest has to be left and there is nothing to the left so this has to be smallest, everything to the right has to be larger than this. So in one line that's the proof for this fact.

(Refer Slide Time 26:06)



So that’s the entire code essentially, you want to find the minimum in a tree x. So while the left of x is not null, just do x is left of x, keep going left of x. When you stop? When left of x is null at that point you will just return x or return the key x or return the element or whatever you want return. So how much time do you take again in this?

(Refer Slide Time 26:10)



So why because of same argument, with every run of this while loop we are going down one step. So we can go down at most the number of the height of the tree and so that’s the maximum time taken. The same procedure can be used in small modification to compute the maximum. We just have to replace the left by right and then we will complete the maximum.

Let’s see how to compute the successor element of a tree? Successor, understand what successor means? Successor means given a particular key after find out the next one. So given x, find the node with the smallest key greater than key of x [Hindi]. Let’s see so there are 2 cases [Hindi] all of you saying key sub tree. [Hindi Conversation]. So there are two cases really so case one is in the right sub tree of x is non-empty, there is something in the right sub tree. So the picture is [Hindi] this is the node [Hindi] successor [Hindi] Right sub tree is non-empty, there is a right sub tree. It exist say [Hindi] so then we know that the key which is larger than this, all the keys in here are larger than this. We know that but why should the successor be lying here? Why can’t the successor be some where else?

(Refer Slide Time 27:58)

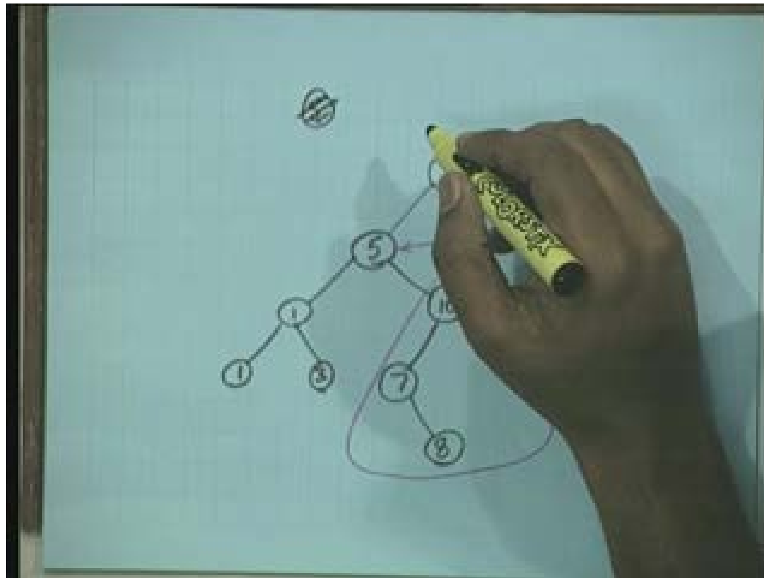
Successor

- Given x, find the node with the smallest key greater than key[x]
- We can distinguish two cases, depending on the right subtree of x
- Case 1
  - right subtree of x is nonempty
  - successor is leftmost node in the right subtree (Why?)
  - this can be done by returning TreeMinimum(right[x])

It is greater than this. So I know that the keys here are larger than 5 but there are other keys which are larger than 5, the parent is one key which is larger than 5. So lots of confusion. So we need to look at it more carefully. So what I am way trying to say, let me draw this picture that was there on the fresh sheet of paper. So I have 5, I will put down 5 here in the center and we have 10 here, we have 7 here and we have 8 here, we have 1, 3. Now I have to find out the successor of this 5 (Refer Slide Time 30:17). I am trying to find out the successor of this node.

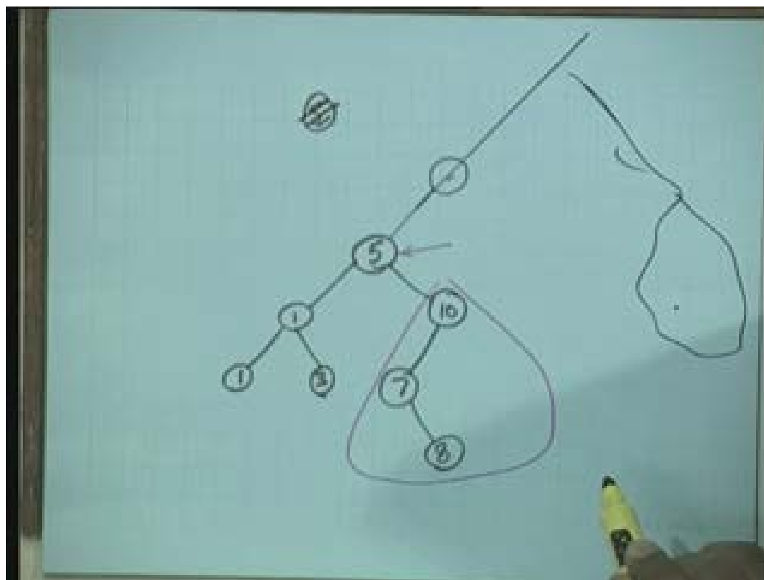
This has the sub tree, now all of you are tempted to say that successor of this guy has to lie in the right sub tree, you want to say that the successor lies here, why? This is not the entire tree, please remember. It could have a parent, the parent is going to be larger than this 5. When it is larger than 5? When it is in the left sub tree. So there are two possibilities, the parent is larger than 5 which means 5 is in the left sub tree of the parent but than this guy will be larger than all of this guys because all of this guys then are in the left sub tree of the parent. The a successor [Hindi] this parent then cannot be the successor.

(Refer Slide Time 31:31)



Why? That is the procedure which you have learnt but why is it that the successor only has to be in this right sub tree but nowhere else if the right sub tree is not null. Conversation between Professor and Student: Refer Slide Time: 31:58). But the next element which is larger than 5 could be let's say this is my entire tree, it could be some where here [Hindi] why not?

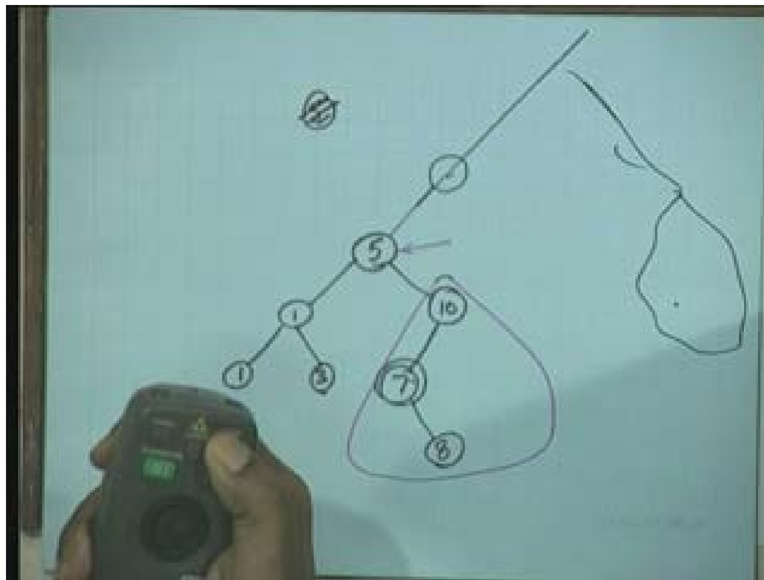
(Refer Slide Time 32:17)



The easiest way to think of this is as follows. Suppose we were searching for some elements slightly larger than 5, we were searching for the successor of 5. The claim is that suppose we were searching for 5, the claim is that we must end up at this node that's a point, whatever

decision we are making, we would end up at this node. Now if the key we were searching for is slightly larger than 5 then that means we are searching for the successor of 5.
Then that means we would second end up with this node and proceed because it is larger than 5 so which means that any successor of 5 really has to lie in this part of the tree. So in this part of the tree the successor is essentially be minimum node because all the keys here are larger than 5. So the minimum key here is the one we are looking for, we have already seen a procedure for completing the minimum which is that keep going left. We will go left, we will go left and we can't go left any further so this becomes the successor of 5.

(Refer Slide Time 34:22)



It remains me the remote that I have at home. So that's what we have to do, if the right sub tree is non empty then we go right one step and keep going left.

(Refer Slide Time 34:24)

Successor

- Given x, find the node with the smallest key greater than key[x]
- We can distinguish two cases, depending on the right subtree of x
- Case 1
  - right subtree of x is nonempty
  - successor is leftmost node in the right subtree (Why?)
  - this can be done by returning TreeMinimum(right[x])

Successor

The other case is in the right sub tree. Suppose I was trying to look for the successor of 3, if I am trying to look for the successor of 3 the right sub tree is empty so where is the successor of 3 now? Look at the parent.

(Refer Slide Time 34:57)

Successor (2)

- Case 2
  - the right subtree of x is empty
  - successor is the lowest ancestor of x whose left child is also an ancestor of x (Why?)

Successor

The procedure is the following and we will see why it is a right procedure. What you are going to do is start going from this node to its parent and then to its grand parent and so on till you reach the node such that this key x that the successor you are looking for is in the left sub tree of that node. So you went up to here but note that this is in the right sub tree of this node. Then you

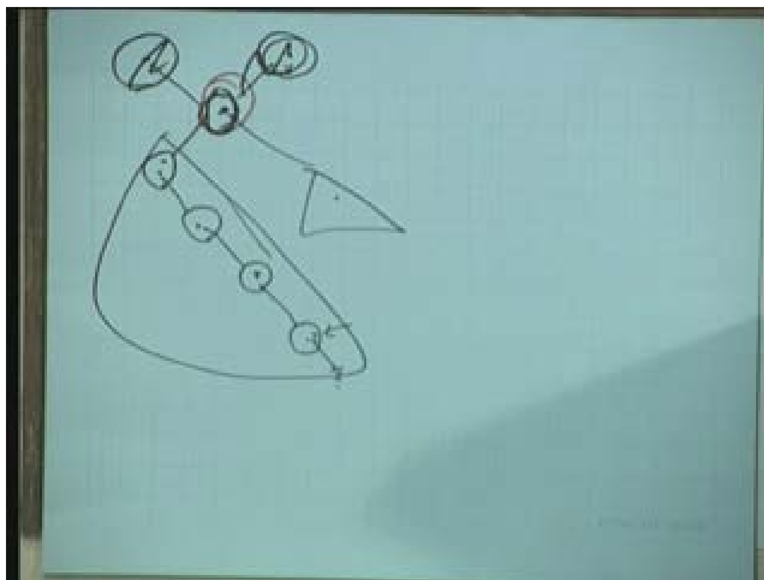
went to its parent and here you found that 3 is in the left sub tree of 5 so we will stop here and this will comes the successor. You understand the procedure but why is that procedure correct?

So recall the procedure is, I am searching for the successor of this node. The right sub tree is null, there is nothing to the right then I go to its parent and I go to its parent and I keep going up to the parent till I reach an ancestor such that this node whose successor we are trying to find is in the left sub tree of the ancestor. So this guy then going to be the successor of x, this node is going to be the successor of this (Refer Slide Time: 37:00). Why this is true?

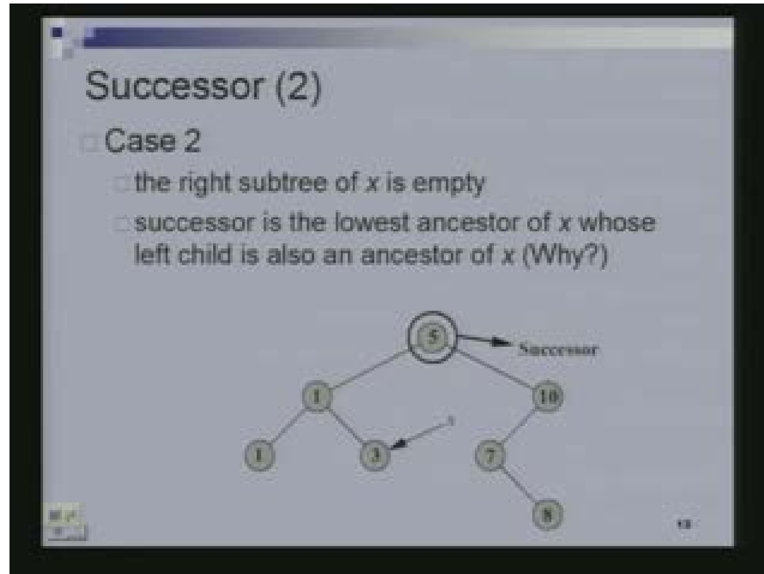
So this node has to be, is it larger than this? No it is smaller because it is in the right sub tree so it is smaller, this is also smaller, this is also smaller, this is larger (Refer Slide Time: 37:23). In fact this this guy is in the largest node in the entire sub tree because how did I find the largest node I come here and keep going right which is exactly what I do, so this is the largest in the entire sub tree and the node following this is then this guy. Yes or no? Can the successor could have been some where else, could it have been to the right of this guy because all of this going to be larger than this, can it be an ancestor of this guy. This ancestor is less than this but this ancestor if it is less than this it is also less than entire thing it's not a successor.

But if you are looking at this ancestor then this ancestor would be less than this and it would be less than entire thing so I was wrong, this ancestor is less than entire thing is not really a successor but this ancestor is greater than this entire thing but it is also greater than this. So this is only the successor so better for confusion but you understand what the arguments are (Refer Slide Time: 38:57).

(Refer Slide Time 39:01)

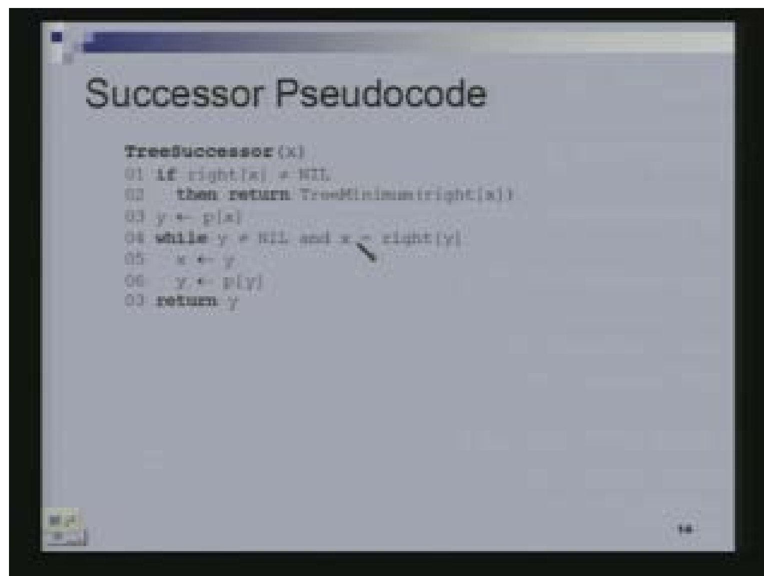


(Refer Slide Time 39:08)



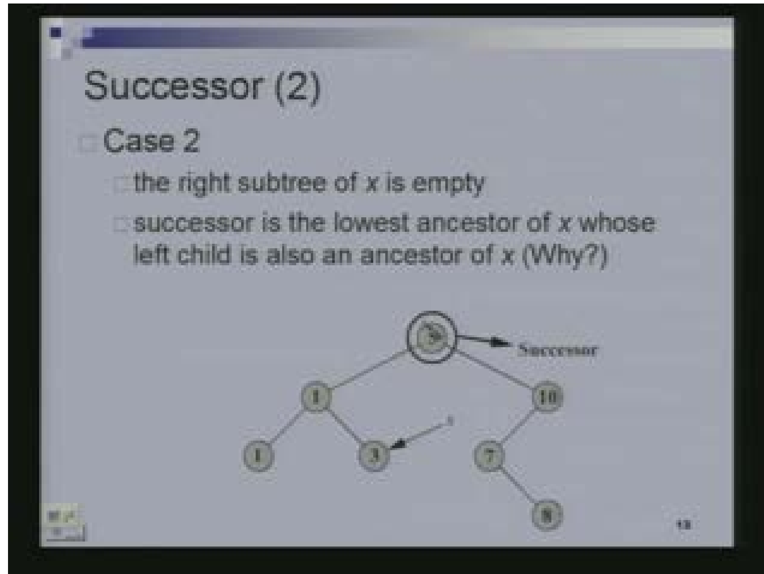
So let's see what code I can write for computing the successor. If this not null then I just return the minimum in the right sub tree otherwise I will come to this point I am looking at the parent, now so y is the parent and what I am doing here? If y is not null and x equals right of y, so when do I have to keep marching till x becomes the left.

(Refer Slide Time 39:56)



So essentially recall I am going up the tree and the first time I essentially I am at the node such that the node is the left child of the parent then I stop. The first time, so let me show you the previous slide. The first time so I keep going up here the first time I come to a node such that the parent such that this node is the left child of the parent then I stop.

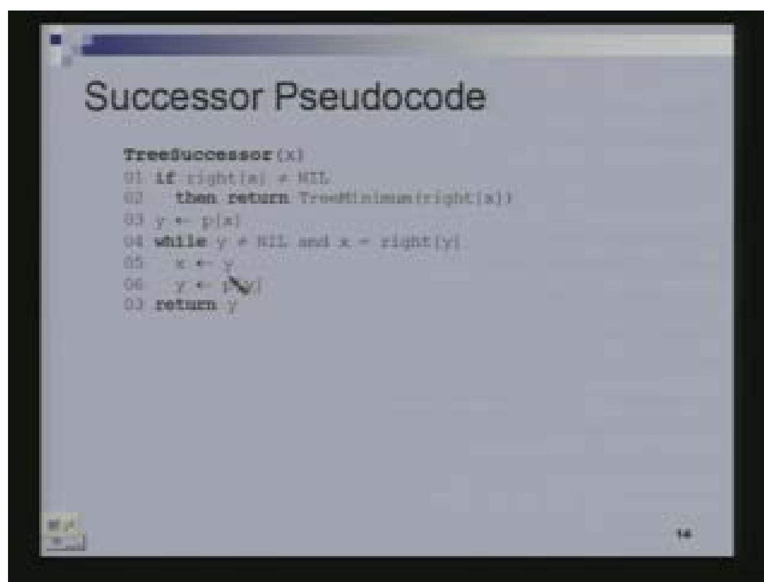
(Refer Slide Time 40:22)



So for instance if I were looking for a successor of 8, I would go up like this and this is the first time the node is a left child of its parent and so I stop it.

So that’s what we are doing here, so initially y is the parent of x and all we are doing is moving the pointers x and y. So y is the parent of x and then we are changing it so that x takes the value y and y takes the value of parent of y, so that it moves one step ahead.

(Refer Slide Time 40:46)

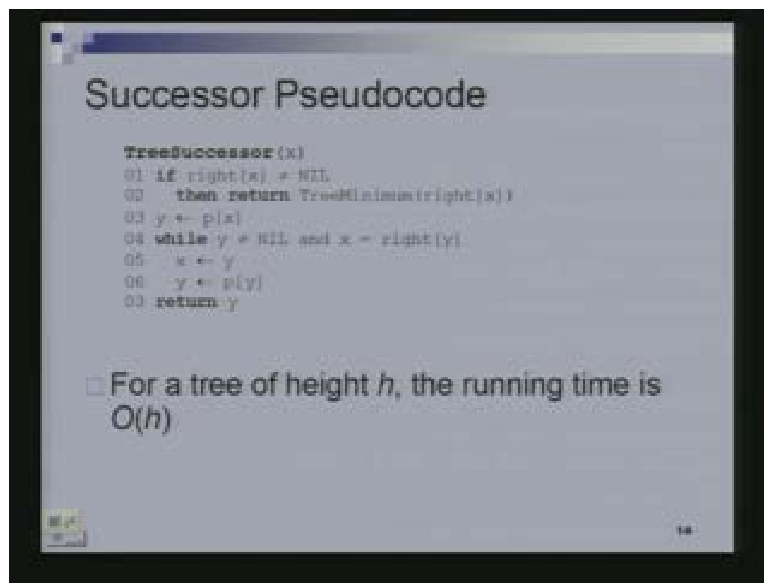


So in some sense these are a pair of pointers which are moving up the tree with y always ahead of x, x y, the x moves and then y moves and x moves and then y moves and so on. When do we

stop? When x is the one which is trailing, y is the one which is head when x is the left child of y. When x is the left child of y we will stop. So essentially we are continuing in the loop till x is the right child of y, when x becomes the left child of y we will get out of the loop or y becomes null that means I have gone beyond the root then we stop and we return. We return y which was the parent node. In case y hits null then we will return y which means null. If we return null which means it could not be found, so you can have a closer look at this code and convince yourself that it is correct.

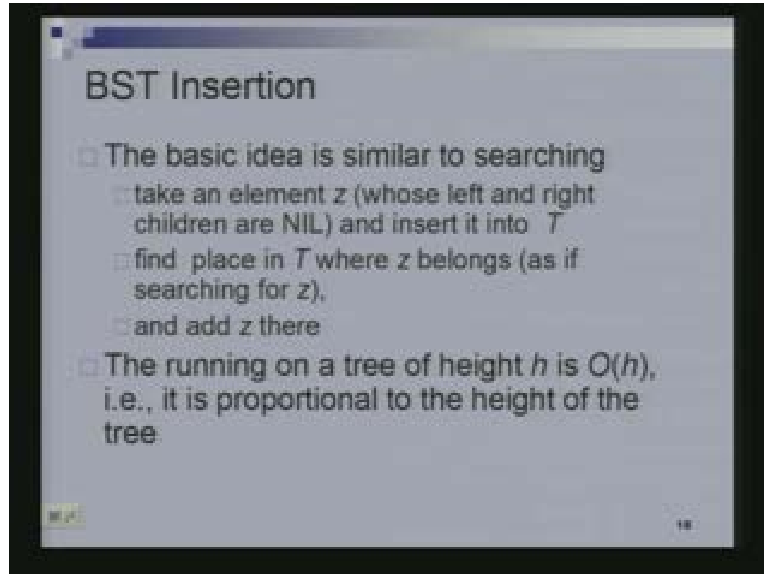
So once again what is the running time of the successor code? Order edge, in each of the cases, in one case what we are doing? We are going down the tree but we only go down as many levels as the height of the tree at most in other case we are going up the tree. So the maximum number of levels we can go up the tree is at most the height of the tree.

(Refer Slide Time 41:44)



So now look at the insertion procedure in a binary search tree. Till now we looked at search, how much time did search take? Order edge, height of the tree. We looked at minimum, we looked at maximum both of them take time proportional to the height of the tree, worst case time and we looked at successor and it took time proportional to the height of the tree. Can you also compute predecessor?

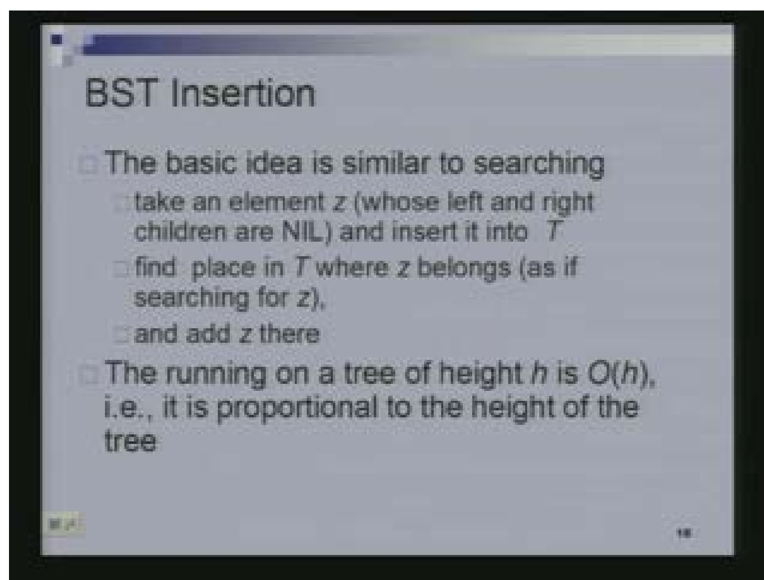
(Refer Slide Time 42:50)



Similar idea, essentially interchange role of right and left, successor may [Hindi Conversation] and we will match up the tree and all of that is the same thing.

Let's look at insertion so that's the other thing we will do today, we will take care of deletion in the next class. So insertion I have a binary search tree and I want to insert an elements so what should I do? Alpha search for the elements.

(Refer Slide Time 44:00)



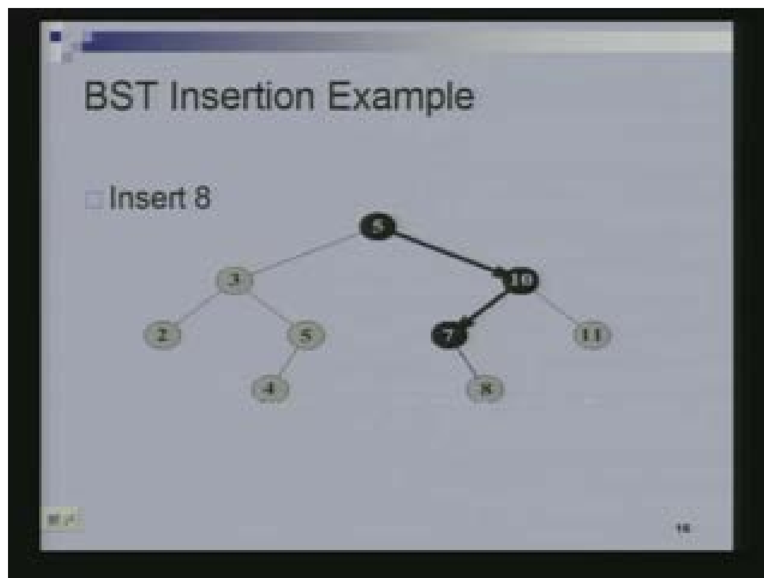
I first find out the place in the tree where the element should be. We are assuming distinct keys, we are assuming that node to elements of the same key. So we first find out where it should be

and then we put it there so that it is actually there. So let's look at an example, this is my binary search tree and I am trying to insert 8. So [Hindi] you recall this 8 is not there in the tree so I fist search for 8 which means I go down.

I come here go right, I go left and then I want to go right but right is null which means 8 is not in the tree, so I put 8 here which means that 8 becomes the right child of 7. At which ever point the search fails because we hit a null pointer and null reference we put the node there that's all insertion is.

How much time does the insertion take them? Order h, which is same as searching essentially [Hindi Conversation]. So now that brings me to that question I had asked at the beginning of the class which is in what order should I been inserting the keys in a binary search tree so that its height… So we said the maximum height of a binary search tree is in or in minus one or one of those things. This is the maximum height of the binary search tree. Suppose I have n keys, in what order should I insert those keys so that I can get a tree of height n – 1? Let's say our keys were 1, 2, 3 up to n. So what order should I insert them, exactly this order or the descending order. So [Hindi Conversation] we will insert 4 at this place and in this manner we will create the chain of this kind and we will get a tree of height n – 1.

(Refer Slide Time 46:03)

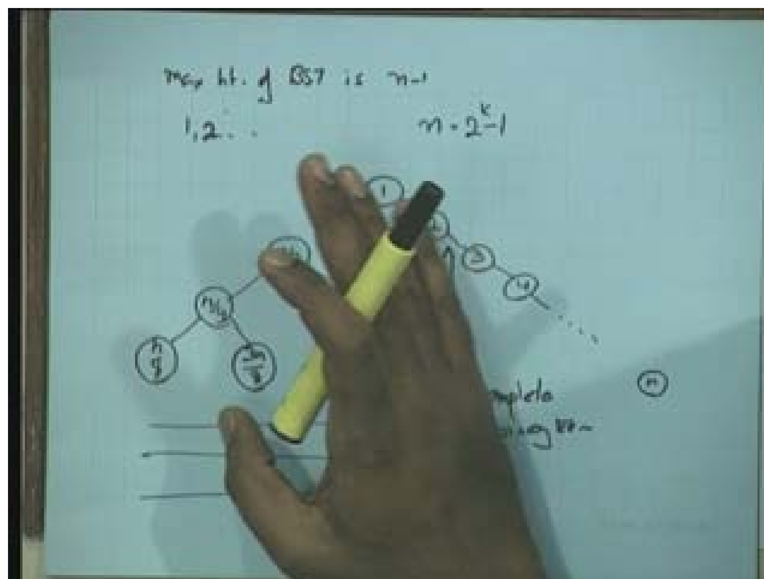


Similar would happen if I had inserted in the descending order. So which order should I insert it so that it can get small height? The first element I should insert is n by 2. Which is the next element I should insert? n by 2 – 1? n by 4 and 3 n by 4 and then I would get something like this and now n by 8 and is it clear that n by 8 will come here? Why this because you will compare and put it here and which is the next one? 3n by 8, suppose I insert 7 n by 8, if I insert 7 n by 8 what happens? So can I insert 7 n by 8 at this point, can I insert 7 n by 8? No harm I can also insert it here.

7 n by 8 will always come here, now I can insert my 3 n by 8 and now I have to insert 5n by 8 which will come here. Essentially I have to go level by level or I have to insert these nodes first and only then I have to insert later nodes otherwise. So in this manner I will end up with what kind of a tree? Height balanced, a complete binary tree I have not told you what height balanced trees are, you don't know what the height balanced tree, you don't even know the full binary trees is, we have learnt only what a complete binary tree. So this we will get is the complete binary tree, complete binary tree provided your n was some power of a 2 or 2 to the k – 1, n has to be of the kind 2 to the k - 1 then you will get a complete binary tree. So in the next class we are going to ask the following question.

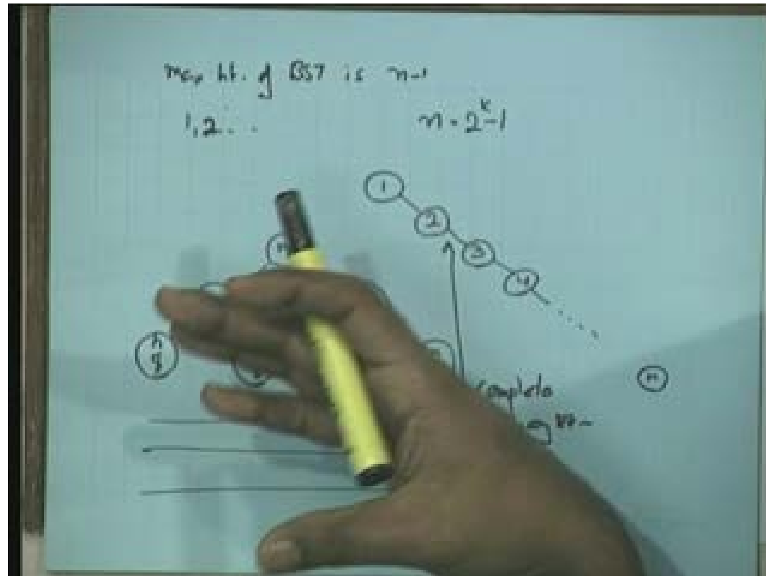
If I was able to take a random permutation of these elements, if I put the elements in an ascending order then I get a bad tree. Why I am calling this a bad tree and this a good tree? Because of the height, this has the huge height, this has a small height why is height such so relevant because so all our operations were depended upon the height of the tree. The smaller the height of the tree, the faster the operation is so you would like to keep the tree height as small as possible.

(Refer Slide Time 50:01)



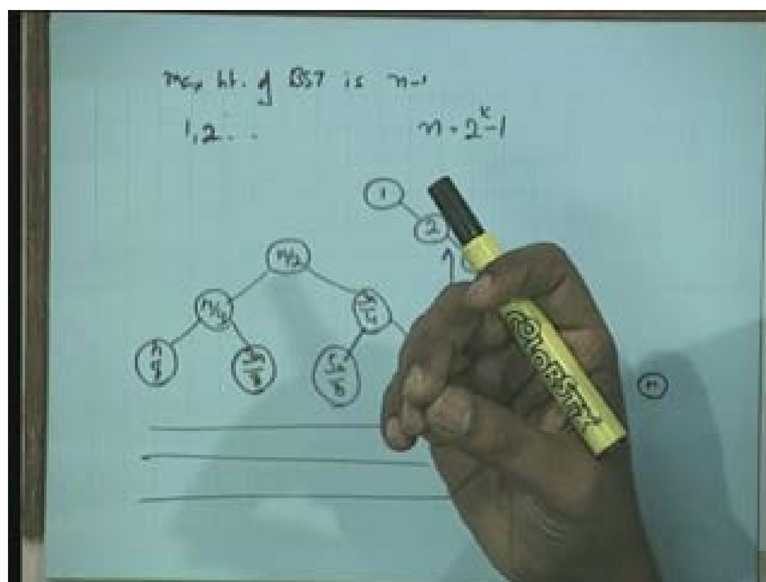
So we don't really want trees of this kind, of course you know with some effort you could figure out in what order should I insert these elements so that I get a small height tree and what is the bad order. Suppose I were to just take a random permutation of one through n, you understand the random permutation. Take one of the permutation there, where n factorial different permutation. Take one of them at random and insert elements in that order. What will be the height of the tree then? Depends on what you or and me? Who does the insertion on the permutation? It will depend up on the permutation so we have to talk of random variable. So the height of the tree now will become a random variable. You understand what a random variable means? Variable which takes many different minimum values so the height of the tree could take one of many different values.

(Refer Slide Time 50:06)



What are the many different values that the height of the tree can take? log n to n, log n to n - 1 or log n + 1 to n – 1, I don’t know exactly which but one of those. log n to n so it will take one of these different values and it will take each of this value with a certain probability because after all I took a random permutation. So what we have to find out? So taking each of these values in the different probabilities so you have to find out what the expected value of these random variables. We will address this kind of a question in the next class.

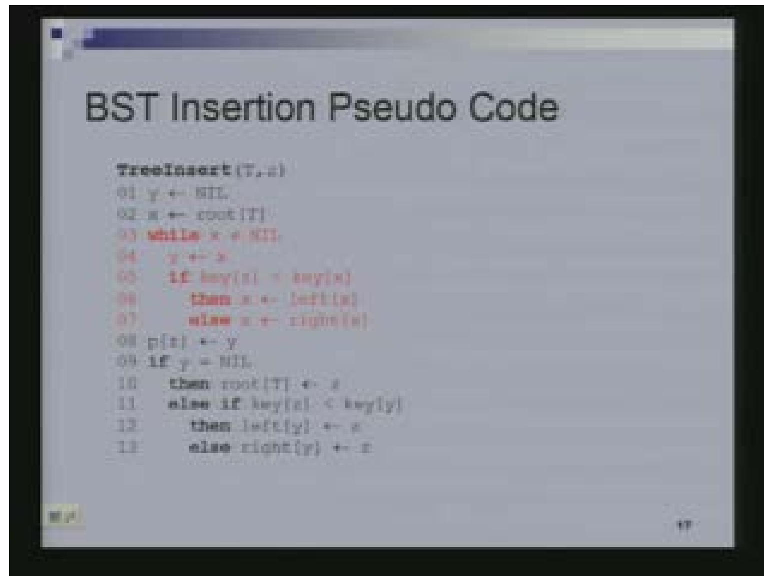
(Refer Slide Time 51:47)



So this can be the code for insertion, so I am given a tree t [Hindi] I will just switch. This can be the code for the insertion, I am trying to insert this node z into a tree t and once again I am going

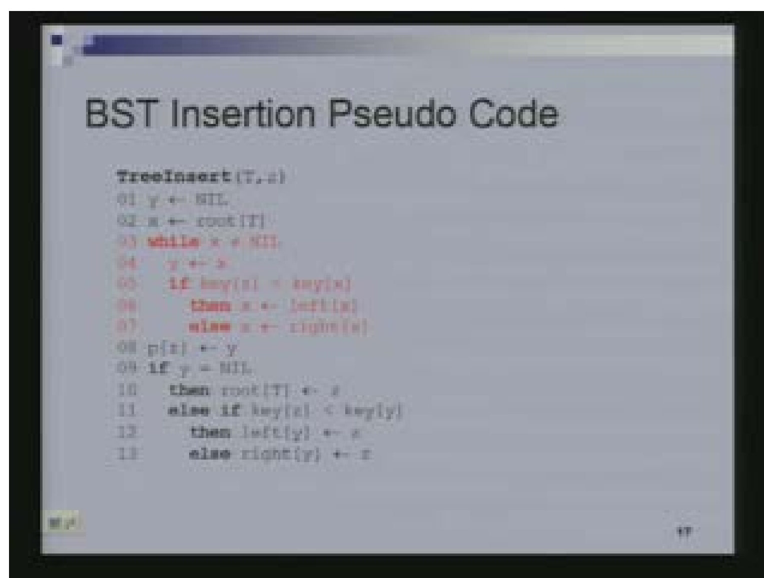
to have two pointers x and y and what is the need for this 2 pointers x and y because recall when I am inserting a node so when I am inserting this node z, I am searching for z the key corresponding to z in the tree.

(Refer Slide Time 52:21)



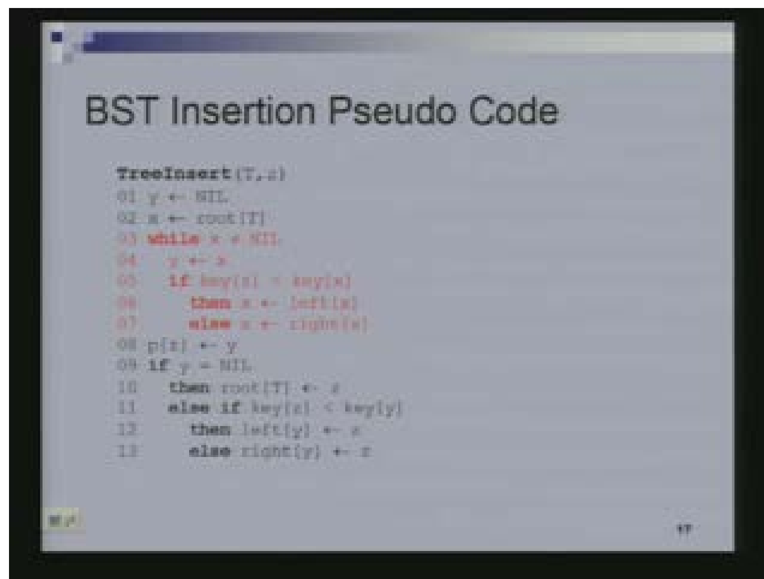
Then I hit this null and so we said we will put z at that place but to put z at that place what we will have to modify? The parent of z [Hindi] right child [Hindi] modify child a left child [Hindi] modify [Hindi] essentially parent pointer parent [Hindi] pointer [Hindi]. We will have two pointers x and y such that now the game is that y will be preceding x.

(Refer Slide Time 53:28)



So these two pointers will keep moving down such that y will always be the parent of x. So that's what we are doing initially y is null, x is the root then y takes the value x and x goes to either left or to the right child of x and we keep doing this till we hit null, till x becomes null when x becomes null y is pointing to the right node. [Hindi Conversation] so y is going to be the parent of new node we are inserting, so parent pointer of z so recall that the node is also have a left child and right child and then parent. So parent of z gets the value y and depending up on whether the key of z is less than the key of y or more than the key of y, if key of z is less than the key of y then z will be a left child of y. Then the left child of y becomes z and if key of z is more than the key of y then the right child of y becomes z.

(Refer Slide Time 54:18)



That's all you have to do but does everyone understand what the need for having these two references. You need to keep track of the parent because that's the one you have to modify so you will go modify that and you will just set it appropriate to the left to the right. So already I discussed this, so in what sequence the insertion should be done, we saw that which should be worst case sequence. So with that we are going to stop our discussion.

(Refer Slide Time 55:20)

BST Insertion: Worst Case

- In what sequence should insertions be made to produce a BST of height n?

A

B

C

D

On binary search tree we will continue this in the next class with the deletion procedure for binary search trees and will also look at this questions that I asked you today but if to insert elements in the random order then what is the time taken to do the insertion and what kind of the tree do you get as the result of that.