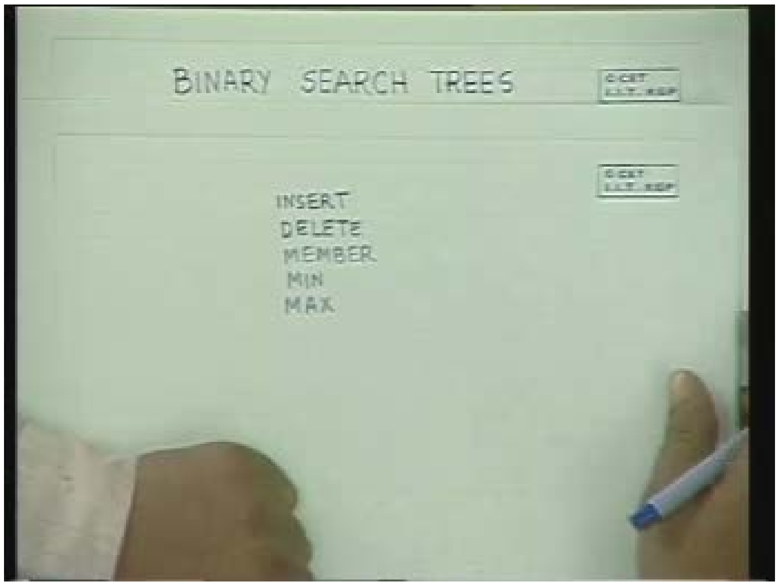
Programming and Data Structure

Dr.P.P.Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture 23
Search Trees II

We shall continue our study of linked lists and binary search trees. The problem that we had in hand in the previous day was that we were given a set of integers with operations insert, delete, find a member, may be get the minimum, get the maximum and if you are representing a set may be you would like to do union, intersection and other operations and you are trying to find out the data structure implementation of this. This will, this problem will come when you are representing a set of numbers, when you are representing a list of numbers, when you are representing a roll list and various other forms you would find some operations, may not be integers may be instead of integers you may have a set of names but all of them can be compared. You know, there is a comparison operation between any two elements instead of these being integers may be you could have a set of strings that is names like you store it in a dictionary or you would store somewhere where given two such elements, you can compare and by that comparison you can say somebody is less than equal to or greater than. So these techniques will be applicable everywhere in all these cases.

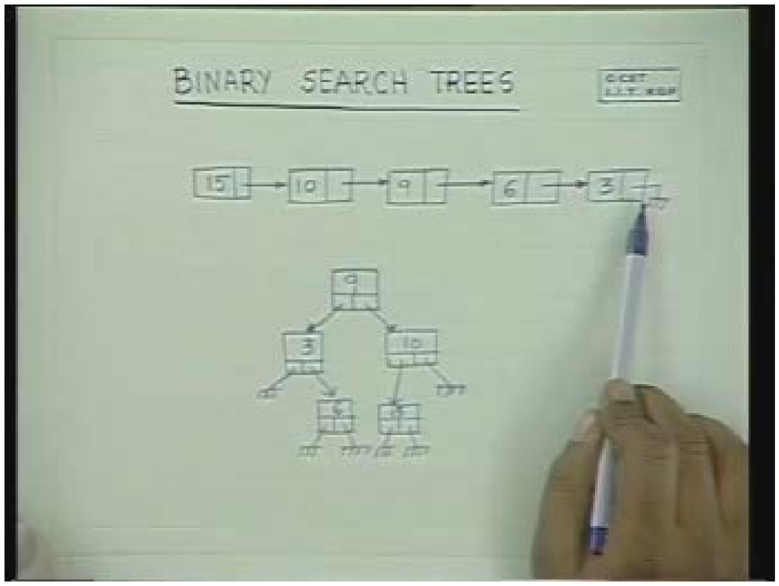
(Refer Slide Time 02:33)


BINARY SEARCH TREES

INSERT
DELETE
MEMBER
MIN
MAX

We saw unordered lists, ordered lists and then we came on to something called binary search trees and we ended up here. We saw that if we order suppose we have got some 5 elements and we ordered them, we can place them in ordered linked list like this where the insert operation would place it in the proper place. The other operations which can scan from the beginning to find it then we saw that instead of having just one pointer which points to the ordering function, we could possibly have two types of links at a note and say those which are less than are on the left and those which are to the right are greater than the element concerned. And identical elements can be stored in the same set by just placing another counter here which indicates the number of times this element occurs, if it is not a set that if it is a list where duplicates will also be considered. And we saw insertion, deletion, etc the basic operation of member would inherently take less comparison than a linked list. This is because a linear linked list would sort from the beginning and would find it somewhere here.

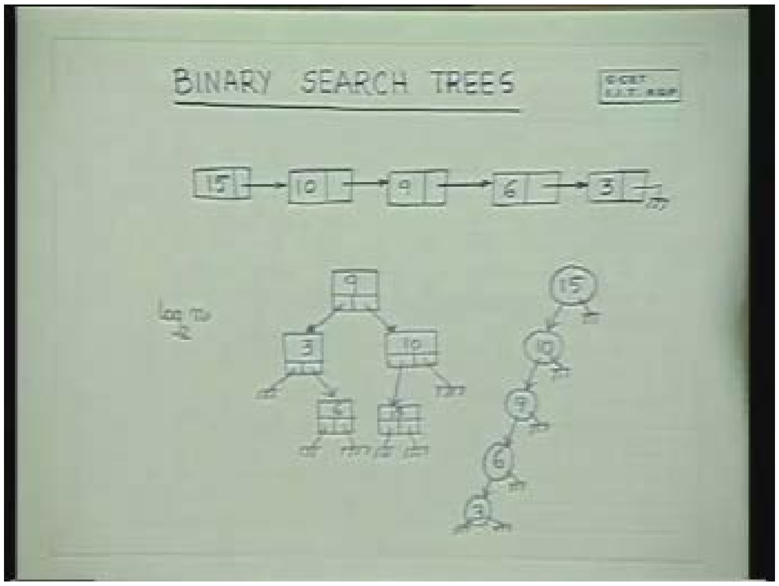
(Refer Slide Time 03:44)



In the worst case it would be length proportional to the length of the list; here it would be proportional to the height of the tree. This is called the binary search tree and the number of comparisons to find an element would be proportional to the height of the tree. Now in the worst case that this tree as we saw in the previous day depends on the order in which the elements come and if the order in which the elements come are say this order then the tree would become 15, the next element would become 10, the next element would become 9, next element would become 6, next element would become 3 and this would actually become in a pathological situation, it would become just like a linked list structure with these pointers being null.

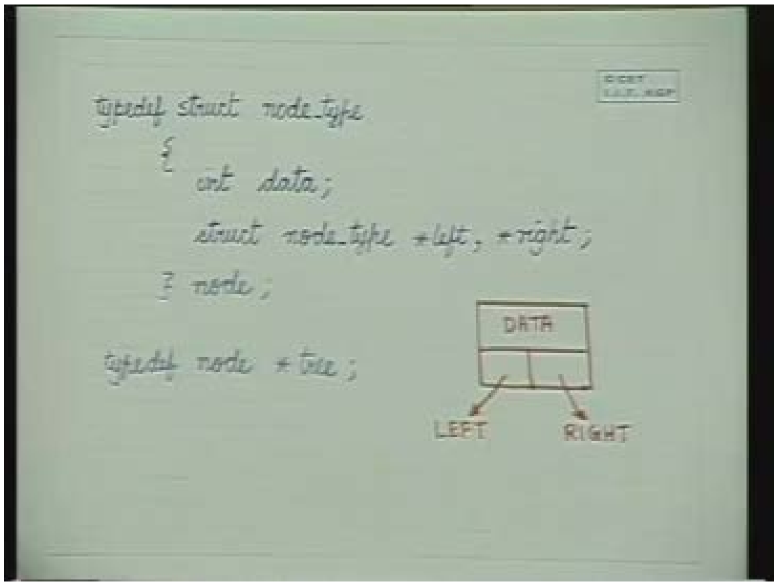
So for a bad situation, this structure would be like this but in other situations you would expect the tree to be of a better quality. You can prove that on an average, if the numbers come from a random distribution then the worst case height would be if n is the number of nodes, you could prove we will not do it in this class. On an average the worst case height would be log n. You can also modify the algorithms to make sure that the height in the worst case is also not more than log n. So binary search trees provides a very interesting way of solving these linked list, these problems where the operations are insert, delete, member, etc, etc. So before going into the technique of minimizing the worst case length, we will quickly go and see what the algorithms for insertion, deletion, find minimum etc would look like in a binary search tree. Here in a linked list we have very clear, reasonably clear about what these algorithms would be. In today's class we will quickly go through what the algorithms will be here. Any questions?

(Refer Slide Time 06:02)



Okay, let's first see what the node structure would be. In a binary search tree, the node diagram is like this, there is a data, there may be other records also but there is a data left pointer and a right pointer and the structure will be like this typedef struct node_type, data node_type left and right, isn't it? There will be a left pointer then a right pointer and a tree is a pointer to such a node. For example from here if a pointer to this node will give you this tree, a pointer to this node will give you this tree, a pointer to this node will give you this tree by default null is also a pointer to a tree, so we define our tree type to be a pointer to such a node. I hope this is okay. Any questions?

(Refer Slide Time 07:03)



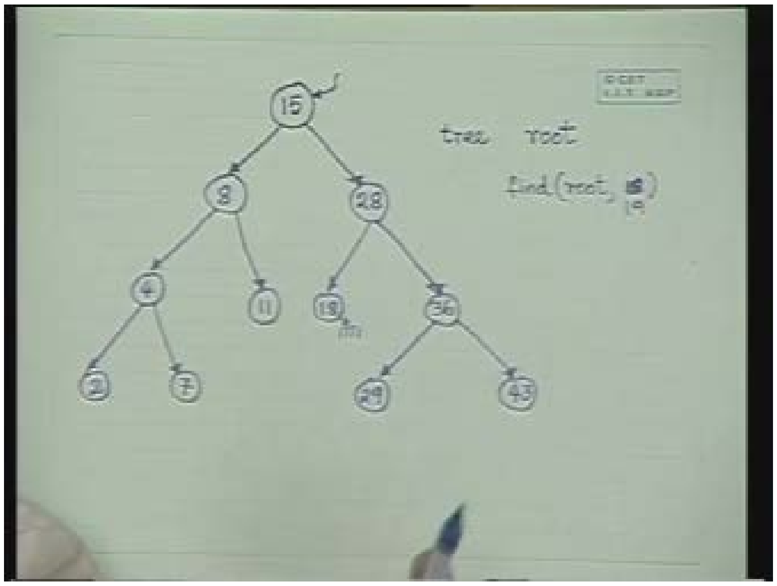
```
typedef struct node_type
    {
        int data;
        struct node_type *left, *right;
    } node;
typedef node *tree;
```

DATA
LEFT
RIGHT

You need not take down notes, this is standard. Any book will tell you. Now suppose our tree at any instant of time is like this. Now let us see how useful this tree is for doing our operations and how we will perform our operations. So we are provided a pointer to the start of the tree that is we are provided a variable of type tree start, the start of the tree is usually called the root of a tree, all right. So we will be provided with the pointer of this type, variable of this type there is a pointer to the node and let us say this and we wanted to find out, we want to do several operations. Suppose we want to do an operation find. Find with this variable root, a value say 18, what will our algorithm be, can anybody suggest? At any node what will the algorithm be? If root pointer dot data we check whether this is the value, this is the value or not, if this is the value then we return this pointer because this is what we want to find. If the pointer is null, we return failure that is it does not exist, all right.

Now there are two more cases left that the pointer is not null and this is not the value, so the value 18 or the value x which ever you want may be less than this or greater than this. If it is less than this then you recursively call it a root pointer dot left, otherwise you recursively call it on root pointer dot right. Clear? Is the algorithm to find an element is clear? Well, I will come to the code also. So the algorithm to find an element and suppose it is not there then what would you return? There are two options, you can return null that is when you have reached the leaf you can return null okay, otherwise you could return the place where it should be put in where suppose this 18, instead of 18 you give 19 then you would look here, this is not null, 19 is greater than this, so you will come here. This is not null, 19 is less than this it will come here, this is not null 19 is greater than 19 so you would come here which is null and you can return null. That is one option. The other option you can return a pointer to this node.

(Refer Slide Time 10:31)

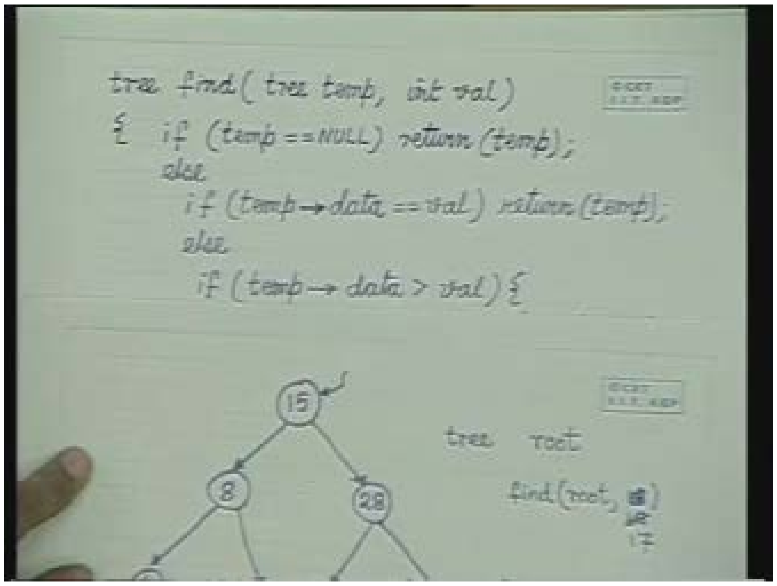


Why you could return a pointer to this node, we will come to it later on. So suppose we are going to return null. Yeah, somebody said insertion, isn't it? If you were to insert 19 where would you insert it? Here, isn't it? So if you find the operation, if the element exists you return a pointer to that element. If the element does not exist, you return a pointer to the parent where it is to be inserted then you could also insert it at that point. So let us have a look at the find operation, all right. Tree find, find returns a pointer to this node, find returns a tree type. Isn't it? Is that understood? Please stop me if you don't understand because I want this part to be absolutely clear. This is the variable which is passed temp, this is the value which I want to find. So let us assume it is here.

if temp is equal to null return temp that is the first thing else if temp pointer data is equal to value then return temp, this part is also clear, temp pointer data. Okay instead of this, you could use start temp with dot data, this is the same thing. Instead of this temp pointer data means the content of the temp. What temp points to which is the content of temp. You could also use the content operation and do, it is the same else if temp pointer dot data is greater than val then what would you do? Return find of temp pointer dot, temp pointer dot data is greater than val that means suppose it was 5 then this is greater than val, so you would search on the left. So the function at this point would be return find temp pointer left, right and if it was null you would have returned null. This is what you would do.

On the other hand if we are interested in, if it is null that is if it does not exist we are interested in returning the parent then you have to do something else that is we have to check here if the left pointer is null. See suppose we are to return 19 then here we would find, suppose it is 17 okay, then so 15 would come here then it would come here, at 18 we have found this condition to be true, temp pointer dot data is greater than val. So if we, if we are interested in just returning null if it does not exist then we would simply call it on. Temp pointer dot left and then you would come back here if null you return but if you are interested in returning this element because it is null, why because you would insert it here. then we will have to make a check that if temp pointer dot left is equal to null then you would return temp otherwise you would continue, so that’s what the case is here.

(Refer Slide Time 14:20)

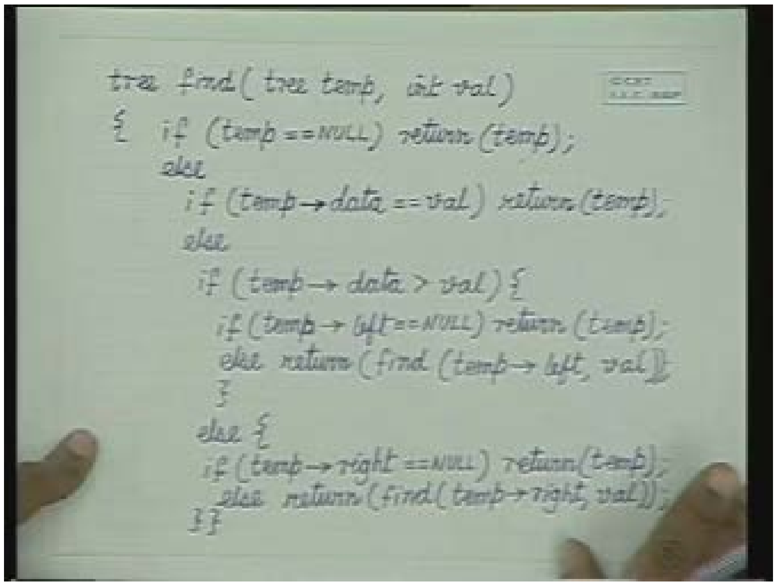


```
tree find( tree temp, int val)
{  if (temp ==NULL) return (temp);
   else
     if (temp->data == val) return (temp);
     else
       if (temp->data > val){
```

15
8
28
tree root
find(root, 17)

If temp pointer equal to null, return temp otherwise that is temp pointer dot left is not null then you would just find it on the left. Is that understood? Any questions? The last case, the last case means temp pointer dot data is less than val, equal to has been checked, greater than has been checked. so if temp pointer dot next is greater than val then here you would simply return right and if it is null and you would like to that node then you will have to make that check, if temp pointer dot right is equal to null return temp else return find if temp right val. so this find routine as it is written here would basically give you, if the element exists it would give you the pointer to that element. If it does not exist that means there will be a null, it will give you the parent. That is what this routine does. And if the start is null, it would return null. So is this function understood, the find function, the most crucial function in a routine like this.

(Refer Slide Time 16:18)



```
tree find( tree temp, int val)
{  if (temp ==NULL) return (temp);
   else
     if (temp→data ==val) return (temp);
     else
       if (temp→ data > val) {
         if (temp→ left==NULL) return (temp);
         else return (find (temp→ left, val));
       }
       else {
         if (temp→right ==NULL) return(temp);
         else return (find( temp→right, val));
}}
```

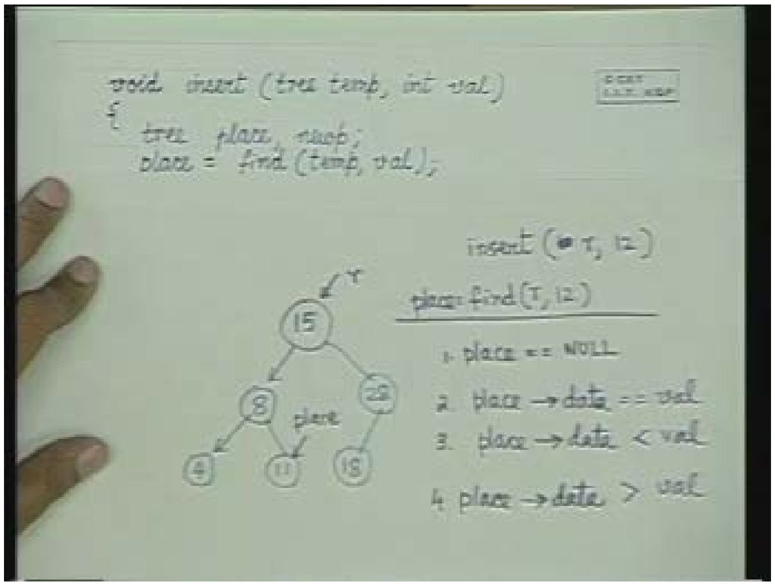
Now if the find has been understood then insert is not difficult. For insert can anybody tell me what is the first thing you would do? Find and get the pointer back. So now you have got a pointer back which says 4 things, basically 3 things, one is if the root is null it will give you null. If whatever it points to, you have to check the value, if that is the value then you have got the element. If that is not the value then you have to depending on the value which you have to insert, you have to insert either to the left or to the right. So suppose you ought to insert 17 then you would first call find 17 and get say place is equal to find of (r, 17) and you would get this pointer. and here we will check, if place dot data is equal to value then either then you will do nothing because it's a set or if you represent all duplicate elements then in the node structure you would have a counter, you would just increment that counter.

If this is not the case then place dot data can be less than val. if place dot data is less than val then this pointer that is place dot data is less than val, so it now has to be inserted this side. Now since you have find return this pointer, this must have been null. Suppose there was an element here 16 then 17 would have to be inserted here. Since it returns this and 17 does not exist then this must be null, right. If this is null then we just malloc an element here and insert it here. The other case that place dot data is greater than null, the same thing you do this side, all right.

So what is the insert routine? Let us have a look at the insert routine. The insert routine will take an argument which is a pointer and value. And suppose you have to insert let's take another example, so it will help us to solve the problem better. Let's take another example; with the same set of data we will take another example. Suppose this is the tree and let us say we give insert 12, this is r, r, 12. Then the first thing that we will do is so find (r, 12). So what will, which pointer will find out 12 will start from here this is not equal to not null, so it will come this side it is less it will come this side. So find (r, 12) so what will, which pointer find dot 12 will start from here? This is not equal to not null, so it will come this side, it is less then it will come this side. Since this is null, it will return this pointer. So when I write place is equal to find dot 12, this will be place.

Now what am I left to check? First I will check if place is null that's all right that means root itself is null and I will have to insert a new element. If place is null then I have to insert a new element, so there can be several cases, one is place is null. Here I will have to create the tree itself that means it does not exist. Second is place dot data is equal to val that means this is the element, I would have tried to insert 11. If I tried to insert 11, I would have returned with this as place and this is one case. The third case would be place dot data is less than val and fourth case is place pointer data is greater than val.

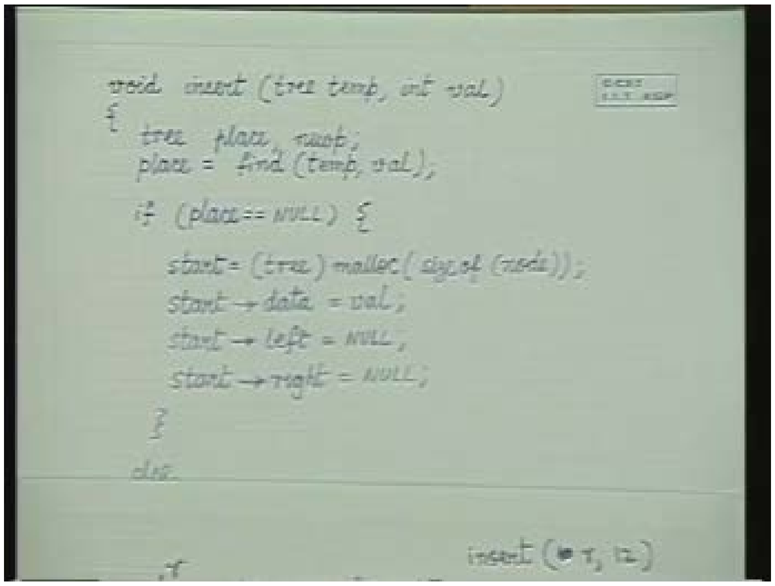
(Refer Slide Time 21:38)



```
void insert (tree temp, int val)
{
  tree place, newp;
  place = find (temp, val);
```

In this case we are here. Now how will we solve this case? That is this case would have a reason if r itself was like this then you would malloc a node, tree malloc this, start dot data is equal to val, start left is null, start right is null. Isn't it and return. There is nothing to return because this just makes the start node, it has inserted it in the… this start is actually a global variable which indicate the start of the tree or root of the tree. So if place was null the root, the tree contain nothing, so we just create the tree else we come to the next case. Let us assume, place dot data is equal to val means we do nothing.

(Refer Slide Time 22:49)



```
void insert (tree temp, int val)
{
  tree place, root;
  place = find (temp, val);

  if (place == NULL) {
      start = (tree) malloc( sizeof (node));
      start -> data = val;
      start -> left = NULL;
      start -> right = NULL;
  }
  else
                                    insert (● T, 12)
```

So this check we don’t make because if this check is there, we just do nothing. Place dot data is say greater than val. Suppose if this is the next case that is else place pointer dot data is greater than val. That is whatever is the data is greater than the value that would have occurred. Here if you place 12 then place pointer dot data is less than val but if you would like to insert say 13 then place would still be here but place pointer dot data would be greater than that. Then where will you insert it? Here, isn’t it? You would insert it here then what is the first step that you would do? You would malloc something here, you would malloc something and assign it to this. You would malloc a node and assign it to this. So that is the first step while left place pointer dot data is greater than val.

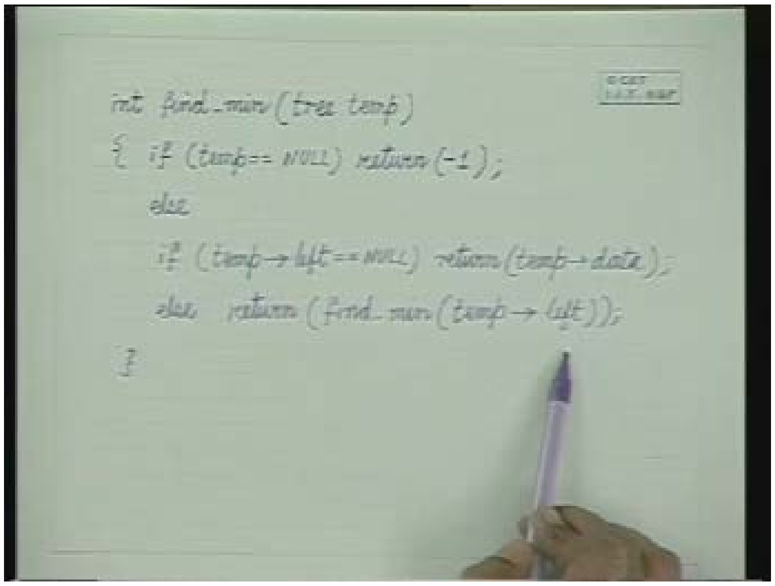
So if val is sorry, that is if val was 10 then this would be the case, not 13, I am sorry. This would be the case when val is say 10. If I want to insert a value 10 then place pointer dot data is 11 and val is 10, so this condition holds. So then I will insert in the left, so I would malloc a node and assign it to the left and I would put place left dot data equal to val, place left dot left is null, place left dot right is null and else if place dot data is less than val, I would repeat the same thing just all of this would be write. I would insert it on the right, just place dot right would be same, place dot right dot data would be val, place dot right dot left would be null and place dot right dot right would be… It’s not dot, its pointer, place pointer right pointer right would be null. so this is how insertion goes and if it is equal, I just dint do anything I came out. I came out because we are trying to represent a set where duplicate elements are not kept.

So in all these cases of a tree, the number of comparisons that I made for finding any element was proportional to the length of the path from that element to the root. Suppose 11, the number of comparisons was proportional to this length from this element to the root. If it was say suppose there was an element here, can you give me a value of an element which exists here. Can a value one be here? 16, can be here, all right. 16 can exist here. Suppose I want to find 16 then the number of comparisons I have to make is this.

For insertion the number of comparisons is the same as the find plus 1 malloc plus 1 more comparison and a malloc. So they take the same, all of them are proportional to the length of the path from the node where the operation is done to the root. Now what else? We were having operations delete will come to, say suppose we want to find out the minimum element then where do we find out the minimum element? You go in left, left, left till you reach null, all right. So you go here left, left, left you have reached null here then this must be the minimum value. The one you go on moving left till you reach some element whose left pointer is null that means that element is the smallest element and that is to anywhere, the smallest element below this is here.

The smallest element below this is here, the smallest element below this is itself. And to find out the smallest or the largest element even then it is proportional to the height of the tree from the smallest element to the root. So, the function for finding the minimum or finding the maximum are very similar and it is very simple actually. It is like this find min, you give a pointer. if temp is equal to null return minus 1 that is it cannot be found that is there is no element at all, so there is no question of a minimum else if temp pointer left is equal to null return temp data.

(Refer Slide Time 28:52)



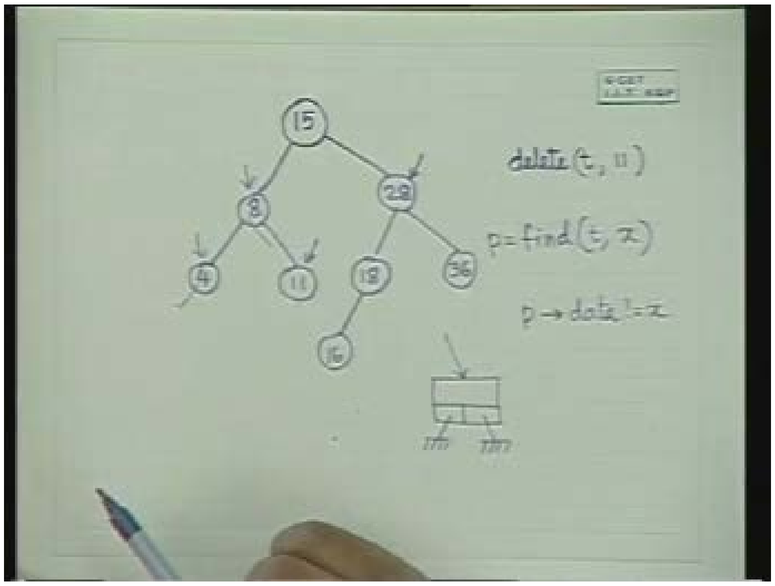
```
int find_min(tree temp)
{ if (temp== NULL) return(-1);
  else
   if (temp->left==NULL) return(temp->data);
   else   return(find_min(temp->left));
}
```

If temp pointer left is equal to null return temp data otherwise you recursively call it on left which will work like this. Here this is not null, temp pointer dot left is not null so you call it here, this is not null, temp pointer left is not null you call it here. This is not null temp pointer, left is null, so temp pointer dot data is printed or returned. So this way you can print out the minimum, maximum you just go all rights.

Now can anybody tell me how I will do a delete operation? That is suppose I give you delete temp 11, you find 11. If find returns you something such that the data is not there then there is nothing to be done, so find operation is very crucial. So to do any delete operation, first do a find operation and then this will return p. you check p pointer data, if p pointer data is not equal to x, forget it.

Now if p pointer dot data is equal to x, what would you do? Let's take the simple case which means to solve these problems; we have to solve it taking simple cases first. Suppose there is no element then find operation would have returned null, so there is no question of deleting. Suppose p pointer data is not equal to val, not equal to x here then also there is nothing to be done but suppose p pointer dot data is equal to x. now let us take some cases. This may be an element like 11 which has got both the pointers null then what would you do? To delete this element you have to, this pointer has to be null, so you have to some how manage and get here, you have to manage to get here, you don't have a back pointer to go here.

(Refer Slide Time 31:28)

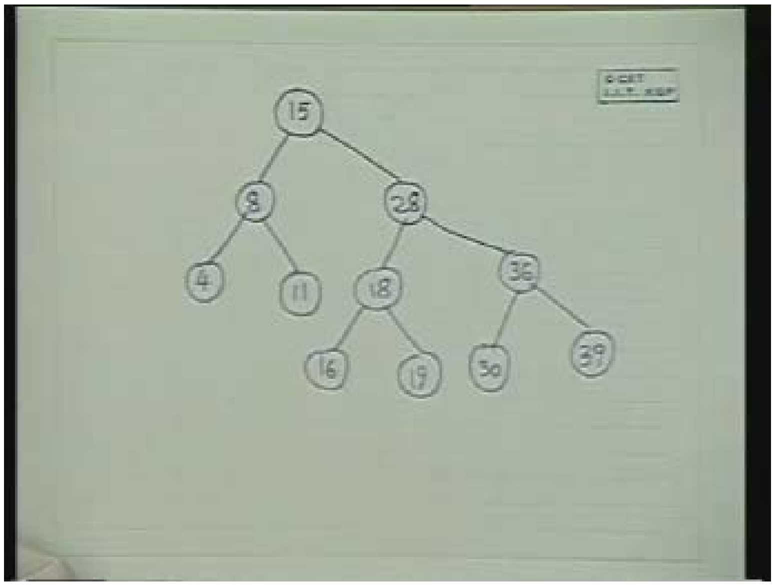


So you have to write a modified find routine to find the parent of the node. So instead of find you have to write a find parent right, instead of a find, a simple find will not lead you to the, so you have to find the parent actually, so instead of doing a find you do a find parent. So to delete a node which has both its children null, we have to find the parent and just nullify this and free this node. So this case is simple. Now suppose I am to delete 28, none of them are null. Can anybody tell me what I have to do? I put this element here. If I will put this element here and then delete this element, no. what? Make 15 point to 36 and make 36 point here. What is the property of this element? Let us look at a cleaner example, will get the… it's not difficult you have to just get the property of that element. To make the case a little more complex and let us say we have to delete 28. What is the property of 28? With respect to this, these elements are greater than 28 and all these elements are less than 28.

So if I delete this, here I have to place an element somehow that I don't have to reconstruct the old tree and I have to reorganize and I can do like this, I can delete the whole thing and then insert all these elements one after another from the root. I could have deleted this and inserted 16, 18 then 19 then 30, 36 so many inserts would have come.

Now among this set, could I insert any element from this set here? If I insert it 19 here it would work, right. From this set can I insert element here, 30. Now can you tell me the general property? On the left side you can replace the maximum element or on the right side you can replace it by the minimum element and the maximum or the minimum element will again have to be deleted.

(Refer Slide Time 34:58)

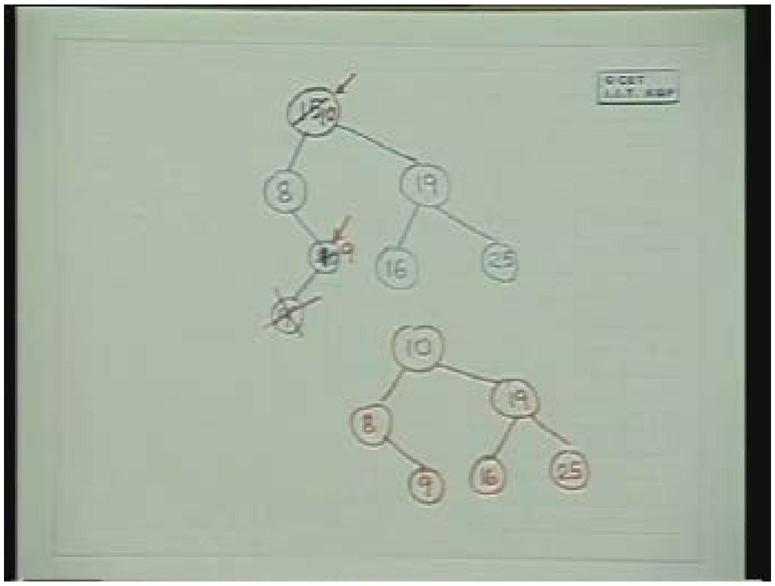


So if you have to delete this element, you find it then we have checked one case where both pointers are null. The other case may be when one pointer is null and the third case which is here is none of the pointers are null. When one of the pointers is null, suppose this side is null or this side is null or these three cases can be tackled in the same way. What you would do is let us take one choice, if the left pointer exists then find the minimum, find the maximum this side. If the left pointer does not exist then go the right side.

If the left pointer exists find the maximum here, so find the maximum you would have come here replace this value here and delete this element. Recursively it would go down, isn't it? Now to delete this element I have reached this case which I can delete easily. Is this understood? So if the left pointer exist, find the maximum on the left side, replace the value there and delete this element. You still only moved along one path. Please note, to find the maximum in the left side again you just moved here. So you will never move more than the length longest path in the tree, you are not seeing all the elements at all.

Now this does not mean in two steps you will be able to solve the problem, I will just take a simple case where you would require more steps. Suppose you have to delete 15 then I would find the left pointer here sorry let there be an element, let this be 10, let there be an element 9. Suppose this is the tree, to delete 15 I would find the maximum on the left which is 10, this would be 10. To delete 10 what would I do? I would find the maximum on the left which is 9 and then to delete 9 since it reaches the leaf I will delete 9 and I will get a structure which is 10 8 9 19 16 25. So using your find and delete and by an exchange and find max and find min, you can easily get the delete routine.

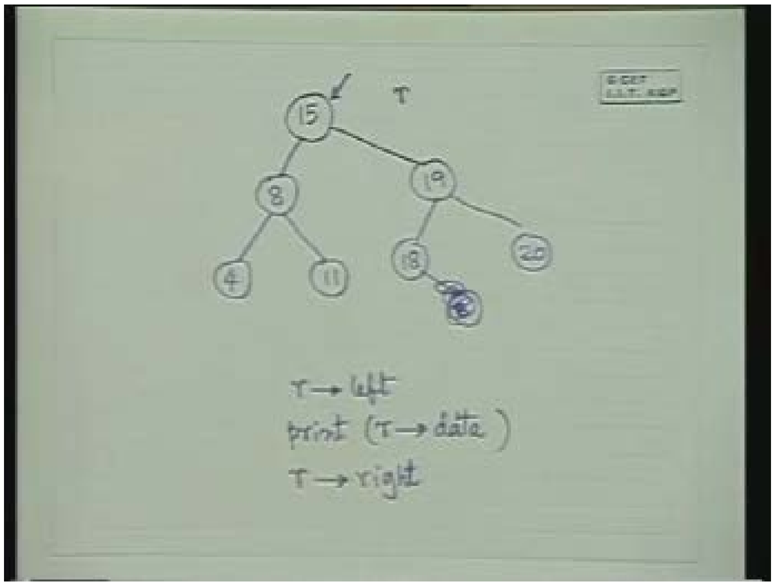
(Refer Slide Time 38:41)



I will leave it to you to write it down because it may not be written down in a page but you can just sit down and work out how to write out the delete routine. But just make a note it is also proportional the number of comparisons and computations will be proportional to the length of the longest path or the elements which are deleted. Only one path you will go across, you will not see more than one path in the node.

Next suppose you have to print these elements in a sorted fashion. Can you tell me what you would do to print these elements in a sorted fashion? You are given this tree pointer here, you would go like this, you would have to print it 4 8 11 15 18 19 20. So if you understand recursion properly, you would do it this way. If this is null you return, if pointer r is null you return, otherwise you recursively call it on r pointer dot left, print r pointer dot data and then again call it recursively r pointer dot right. I will leave it to you to just work it out and we will see it in the next class. We will see two things, one is how to balance the tree so that the worst case height is always log and we will see other functions in the search trees. Search trees are very important in data structures and they will come to place in many examples.

(Refer Slide Time 40:36)



So once we are clear about search trees, we will go back to all our examples in c where and how they will be useful.