Programming and Data Structures
Prof. N. S. Narayanaswamy
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

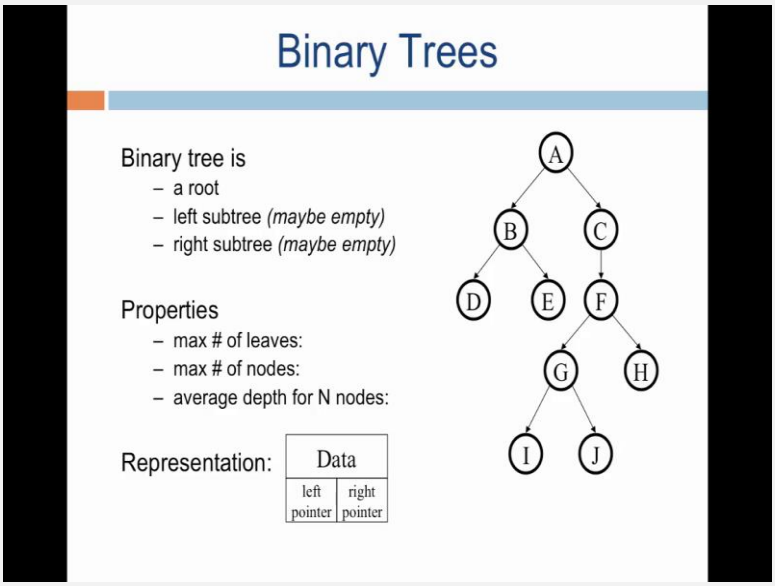
Lecture - 17
Binary Search Trees

Hello, welcome to this last state of lecture in the programming data structures course, and in this last lecture we will looked at two advance data structures that are extremely useful. These two data structures are based upon the non leaner data structures that we studied in weak number 6 of this course and both these data structures that we have going to talk about have extensive applications, we are going to look at binary search trees and heaps.

So, in the first lecture this week we are going to look at binary search trees and in the subsequent lecture we are going to look at the heap data structures, which is used to maintain priority queues and so on and so forth. So, the difference between this set of lectures and the earlier lectures is that the programs that I have typically been showing you are going to be a part of the presentation itself, and I am not going to show you any executable C code or I am not going to show you any execution and compilation steps.

So, I am just assuming that, because we have looked at how binary trees are implemented in general, trees are implemented; once the abstract data type for the binary search trees and abstract data type for heaps are clear the implementation is just adding additional features to your already existing implementation on trees. So, let us just go to the presentation and explore the world of binary search trees and essentially we will look at the different methods that are associated with binary search trees.

(Refer Slide Time: 01:48)

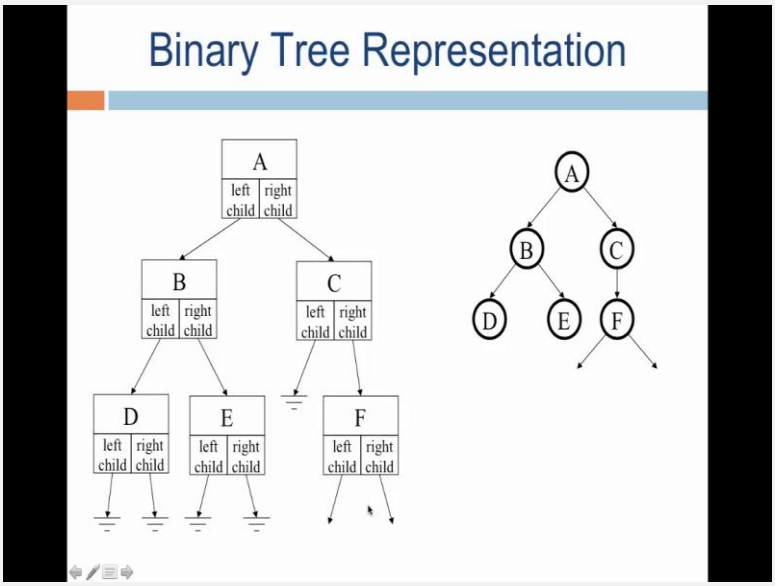


So, we going to talk about binary search trees and let us just recall what binary trees are... A binary tree is essentially a triple, it is recursively defined, it has a root and it has a left sub tree and a right sub tree. In this example for example, A is a root of this binary tree and this tree is the left sub tree and the tree for which C is the root is the right sub tree. It is quite possible that the left sub tree or right sub tree or both may be empty, observe that C has only one child and depending on the implementation, if this is implemented as a right sub tree, then it is left sub tree is empty.

If you look at a leaf node like D, observe that it has neither left sub tree nor a right sub tree. The properties of binary trees are one is interested in binary trees with a maximum number of leaves, one is interested in binary trees with a maximum number of nodes, we are interested in the average depth for N nodes. So, for the example if the binary tree has N nodes, in this case whatever is the number of nodes here you can see with the average depth seems to be something between 3 and 4.

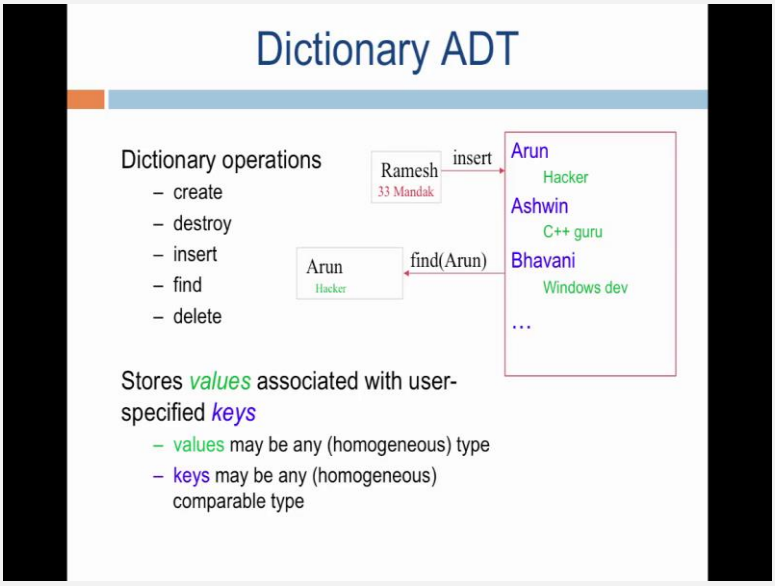
The typical representation of a node in the binary tree which is what you put in your node header file is there is a data item and then there is a pointer, the pointer is going to point to this object itself, the left pointer and the right pointer to your object of this same type.

(Refer Slide Time: 03:28)



So, in other word this is what a binary tree representation is going to look like, here is a node we have seen the code for such things before, A is the key here, the left child points to the node that contains the value B, the right child points to the nodes that contains the value C and so on, and here is a notation for a null pointer, where the left child is not pointing to anything. So, this is the representation of the binary tree that we have already seen.

(Refer Slide Time: 03:58)



Now, let us look at an abstract data type that we have not seen so far which is called the dictionary abstract data type. The dictionary abstract data type is a very important abstract data type, it is extensively used and it is extensively used almost anywhere, where you want to maintain a table or a set of words and you want to repeatedly query this table, the table is the abstraction. The data structure, the abstract data type is the dictionary data type which means that the keys have an order.

For example, we all know that Arun is lexicographically smaller than Ashwin. In other words, if you maintain a dictionary Arun would precede Ashwini and Ashwini would precede Bhavani and so on. And this is exactly the dictionary order in which we view words. Now, the typical dictionary operations in the abstract data type are that you want to create a dictionary, you want to destroy a dictionary and you want to insert a key into the dictionary, you want to find the key if it is presented in a dictionary and you to want delete a key from the dictionary.

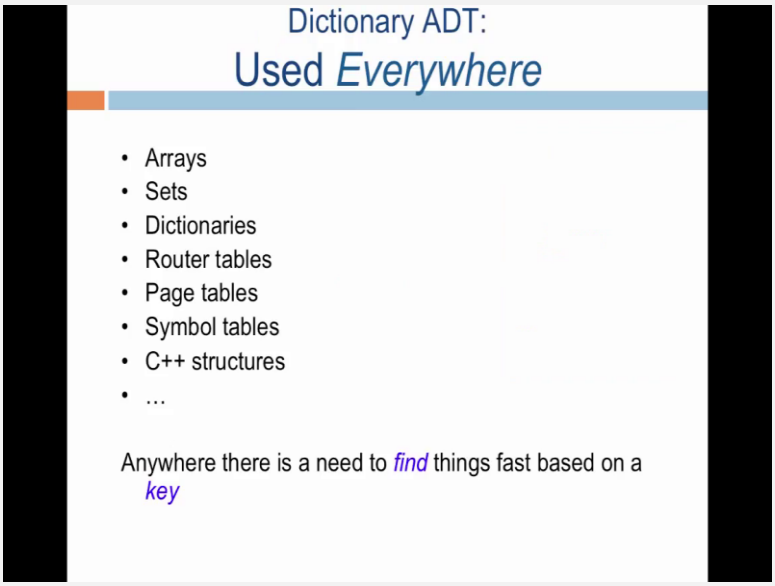
These are captured by this pictorial representation, so if you wanted to insert the key Ramesh with the attributes which in this case terms out to be may be the room number of Ramesh in a certain hostel. In this case, Arun has a certain attribute that he is a hacker, Ashwin is a C plus plus guru and so on and so forth. So, you would insert the keys Arun, Ashwin and Bhavani and then you would insert Ramesh in to it and this is an additional attribute with Ramesh.

Now, when you find Arun, when the query find Arun is made to this dictionary, the response is the key that has been found with any other attribute that is been that is present. In this case, because Arun has an attribute hacker, the response of the find Arun function is to return the key Arun along with some values that are associated with the particular key. So, the values could be any homogeneous type, so which means that you should be able to, they should all be of the same type.

For example, 33 Mandak is a string, C plus plus guru is a string, hacker is a string, so the value associated with a different keys are all of the same type. Similarly, keys are also of a homogenous type, the additional attribute is that the keys must be comparable, meaning that in this case the lexicographic ordering is important. Therefore, when you think of a dictionary abstract data type, you think of keys, the keys must come from a single type, additional property of the type is that any two keys must have a clear comparison of

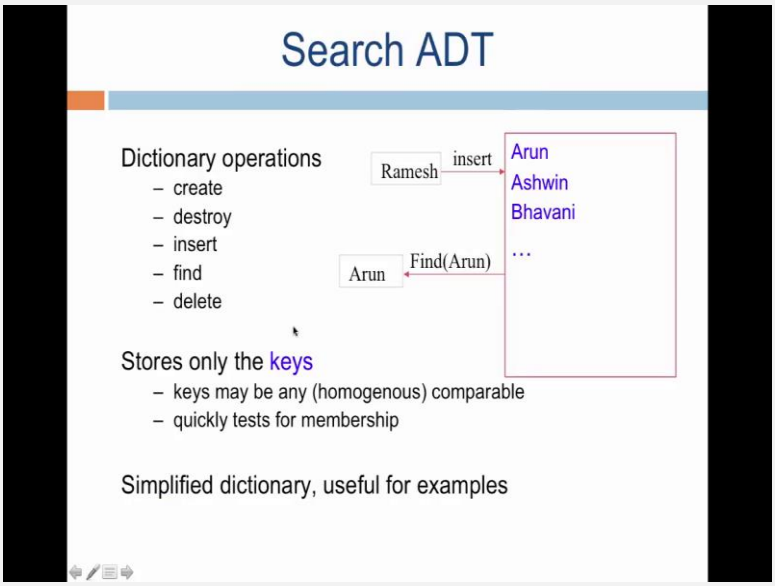
which is bigger and which is smaller. So, these are the methods associated with the dictionary data type and observe here that we have really not looking at, at the moment how to implement it. We just saying these are the wish list functions the list of functions that we wish to implement with it is dictionary data type.

(Refer Slide Time: 07:14)



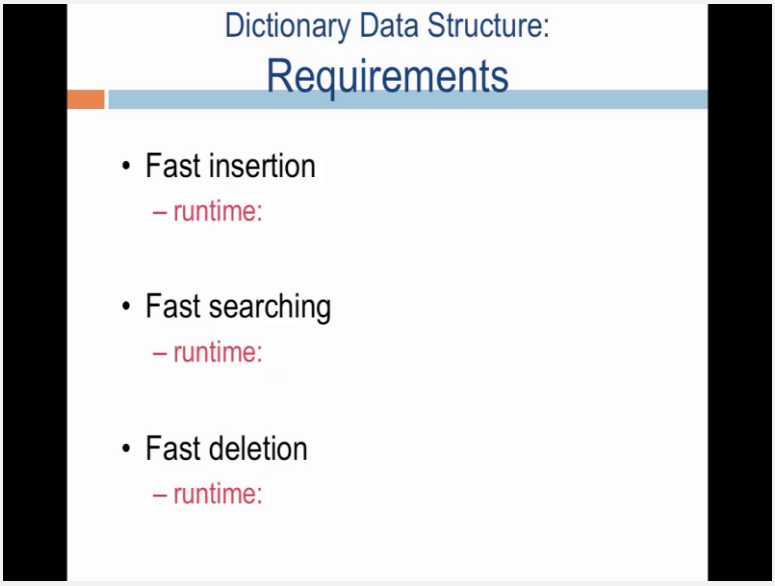
Like I said earlier, the dictionary abstract data type is used extensively anywhere you want to find things fast based on a key, symbol tables. So, if you are writing a compiler, then you might have studied symbol tables and you will definitely use symbol tables. Router tables are like dictionaries where the table maintains network addresses dictionaries exactly word dictionaries that we are all familiar with and so on and so forth.

(Refer Slide Time: 07:45)



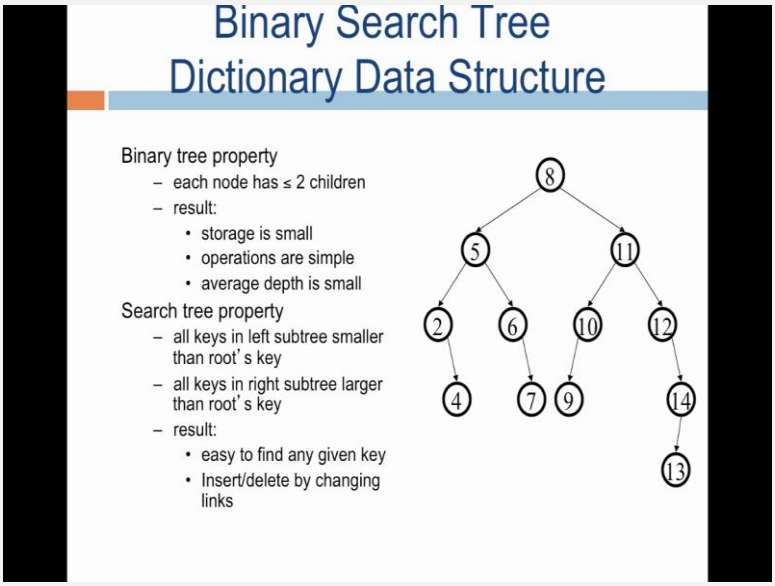
Now, the rest of the lecture we are going to look at how to implement this dictionary search abstract data type, we are going to look at how to store keys, so that we can extract them quickly. Observe that here I have removed all the attributes, I will just put in exactly the keys that I want to store, the attributes can be stored in some other method and by using some other approach. Here we are going to focus only on how to store these keys, so that we can extract them fast.

(Refer Slide Time: 08:17)



So, therefore we want a fast insertion, we want a fast query time, fast search time and a fast deletion time, these are our requirements. So, those we have the methods and we want these operations to be done as quickly as possible in an implementation of the dictionary abstract data type.

(Refer Slide Time: 08:34)



So, here is the binary tree data type or a modification of the binary tree data type that we will used to implement our abstract data type called the dictionary which we saw for a few minutes just now. So, let us move away from the dictionary data structure and just focus on the binary search tree. So, what is a binary search tree? A binary search tree is first of all is a data structures. Secondly, it has two properties, it has a binary tree property and it has the search tree property.

I like to recall again your attention that we are going to use the binary search tree to implement our dictionary abstract data types. So, therefore from now on whatever we discuss is going to only be a discussion on the binary search tree, we are going to define what a binary search tree is, we are going to understand this binary search tree in a very clear way. The binary search tree whenever you think of a binary search tree, you think of two properties associated with a binary search tree.

One is called the binary tree property which is a structural property and the second one is a search tree property. Let us just go through the binary tree property, the tree structure is the binary tree as we saw in the earlier part of this lecture. Every node has at most two

children, it has the binary search tree, it has a definite root and every node has upper bounded by at most two children. For example, 5 has both it is left child and right child, 2 on the other hand has only it is right child and the node containing 10 has only it is left child and 7 is a leaf node it has neither child.

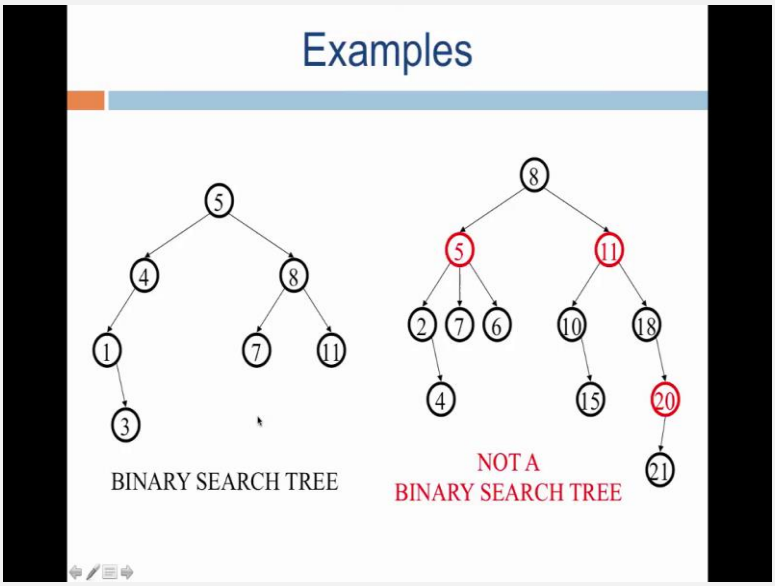
The advantages of our implementation is that we will use storage in a very efficient way, the implementation operations will all be simple, there would be an extensions of the binary tree operations that we have already seen. And however, our wish list is that if you take the average depth of a node what do I mean the average depth of the node, for example if you take the depth of every node, add it up throughout, add it up that is take the depth of all the nodes, take the sum and then average it over all the nodes, this must be as small as possible, this is a wish list.

So, we will really not discuss how to ensure that the average depth is small, but we will definitely point you at the end of this lecture to different methods which are there in the literature that you can definitely explore and implement it. The additional property is the search tree property, the search property has the following features that if you look at any node, here if I look at this root node, the key here is 8 and all the keys that are in the nodes which are to the left sub tree of at this particular node are of value smaller than 8 and all the values which are in the nodes which are to the right are greater than the value here.

So, this is the property and this property holds that every nodes. For example, if you look at 6 all the nodes which are to the left of 6 of course, that is an empty set or smaller than 6, it is an empty set. So, it is find the statement is true. All the nodes which are to the right of this particular node that is here 7 is more than 6, same as towards 5, all the nodes to the left of 5, there is 2 and 4 are smaller than 5 and all the nodes to the right of 5 are larger than 5, that is 6 and 7. This property is respected at every node this is exactly a binary search tree.

Therefore, whenever you think of a binary search tree, you have to think clearly of these two properties, the structure of the binary tree that is a structure property and the order among the keys that is a second property.

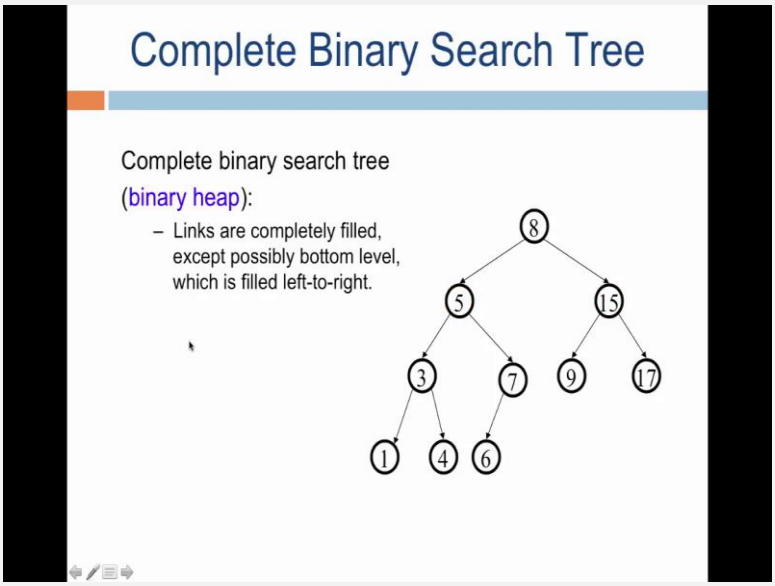
(Refer Slide Time: 12:16)



Let us do some quick examples, so let us do a couple of quick examples. So, this as you can see is a binary search tree and the way to do this is that you have to look at the tree structure. Yes, it is a binary tree, every node has at most two children, there is a well defined right child, there is a well defined left child, no right child and so on and so forth. So, now let us look at the search property, all the keys here are less than 5, all the keys here are all more than 5, at every node you can verify the properties.

As you can see all the keys to the left of 4 are 1 and 3 and they are smaller than 4, all the keys to the right of 5 are larger than 5 and so on. On the other hand, here you note that this is not a binary search tree, because it is not a binary tree. We do not even worry about whether one can meaningfully answer the query about whether the search property is there or not, it is not a binary tree and therefore, it is not a binary search tree. So, therefore the first thing is that every binary search tree is a binary tree.

(Refer Slide Time: 13:41)

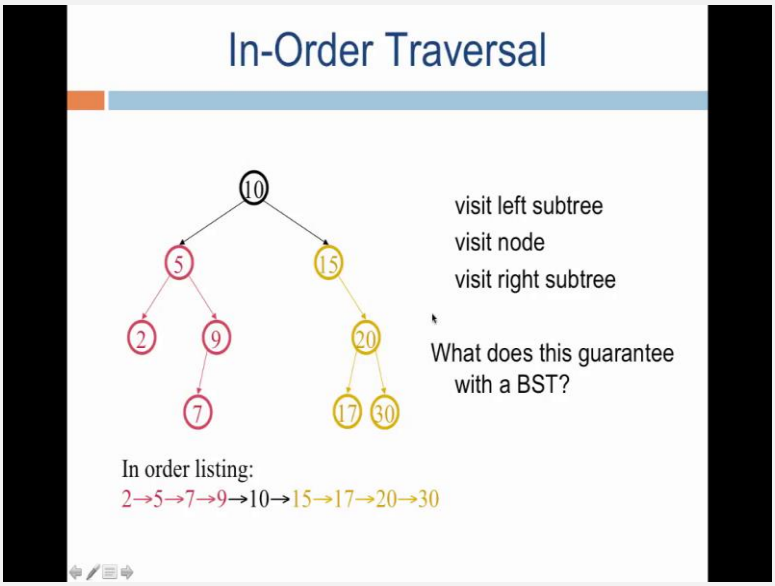


Now, let us look at the best binary search trees that we could have. Now, what do I mean by best binary search trees? They must be binary search trees and their average depth must be as small as possible, ((Refer Time: 13:54)) let us go to the previous example, observe that there are very long paths in this binary search tree. For example, there are 4 plus 3, 7 keys here and observe that it would have been better if 3 had been placed here, 1 was to the left child of 3 and 4 was here.

It would have a binary search tree which was less deeper or shallow over than the binary search tree, I hope the example was clear. This binary search tree is a binary search tree which is good; however, we want to have what binary search trees which are has shallow over as possible. In this case, observe that observe the mouse point there if I moved 3 here to this place and made 4 the right child of 3, then you would have add a binary search tree which is full or all it is a binary search tree of the leaves depth for 7 keys.

Complete binary search tree is what we wish for, it is the links are completely filled, exit possibly at the last level which also is organized from the left most child onwards to the right. So, this is something that you must be already familiar with from the previous lectures, so let us not spend too much time on this. So, this is the complete binary search tree, it is also called a binary heap.

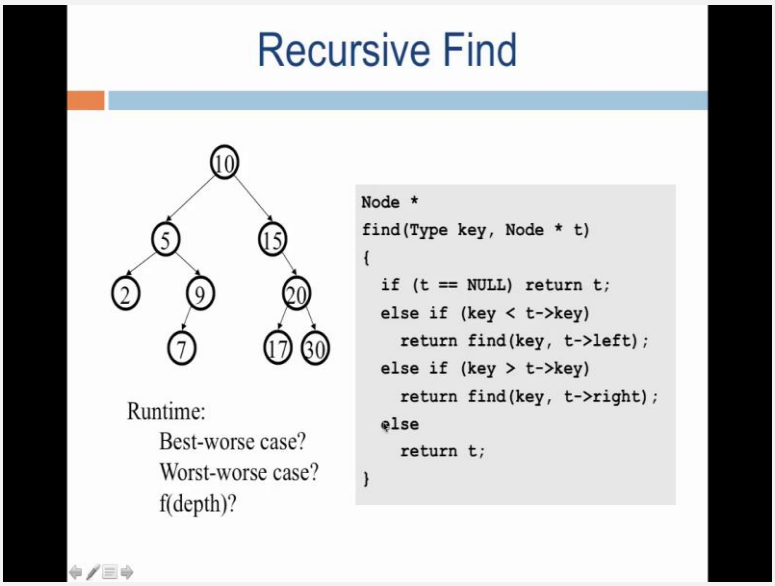
(Refer Slide Time: 15:27)



Now, let us just look at the inorder traversals of binary search, we have different traversals, so we all know by now what the inorder traversal of binary tree is. So, observe that the inorder traversal of a binary tree is it is a binary tree, so you recursively list the left sub tree, then you list the parent node, then the right sub tree. So, let us do this now, I list 2, then 5 then 7 then 9 then 10 then 15, because 15 does not have a left child, so I explore.

Find out this is a null set it does not an address, it is nothing to be printed. So, then I print 15, then I go here prints 17, 20 and 30. So, observer the nice property it is worthwhile trying to implement yourself and verify this from binary search trees, check that the output of an inorder traversal of a binary search tree is always a listing in the increasing order. This immediately tells you that the smallest key in a binary search tree is a left most node and the largest key is the right most node, we will come to this in a minute.

(Refer Slide Time: 16:41)

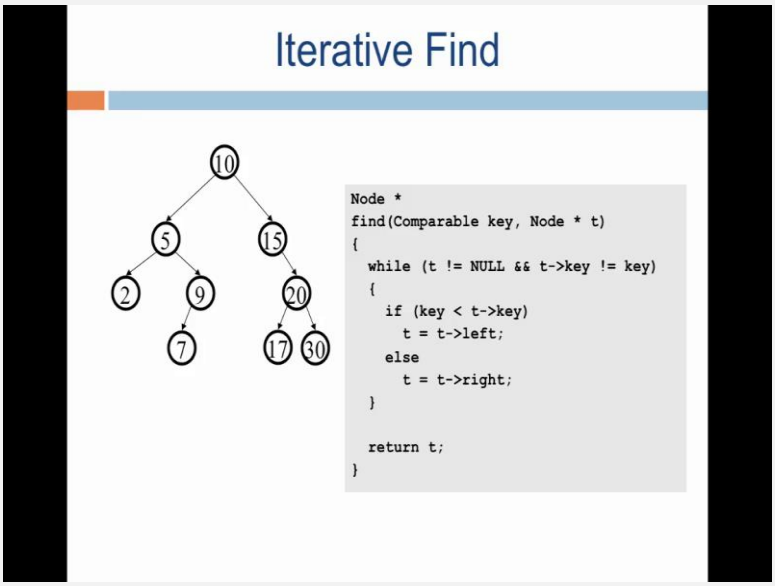


But, before that let us look at how to find whether a given key is present in a binary search tree or not and I suppose to the earlier lectures, in this lecture I am putting out questions that you can think about what is the best running time possible, what is the worst running time possible and how is this connected to the depth of the binary search tree.

So, what is the find function as you can recall from the dictionary ADT, the find function takes an argument which is a key and then pointer to the tree and returns a pointer to the node that contains the key and if the return value is null, it means the key is not present. So, let us look at the piece of code, if t is null then you return t, saying that essentially saying that you have not found the key. Otherwise, you check if the key which is the argument that has been passing to this function is smaller than t pointer at key.

In this case let us say t was pointing to this node, then you will compare key with 10, if it is smaller than 10, you recursively find the key here in the left sub tree that is what this one says return find key in the left sub tree, otherwise you find the key in the right sub tree, if it is larger. Otherwise, if it is neither less than nor greater than, then it must be equal, there is no other possibility therefore you return a pointer to the node at which you are inspecting. So, this is the recursive find.

(Refer Slide Time: 18:19)



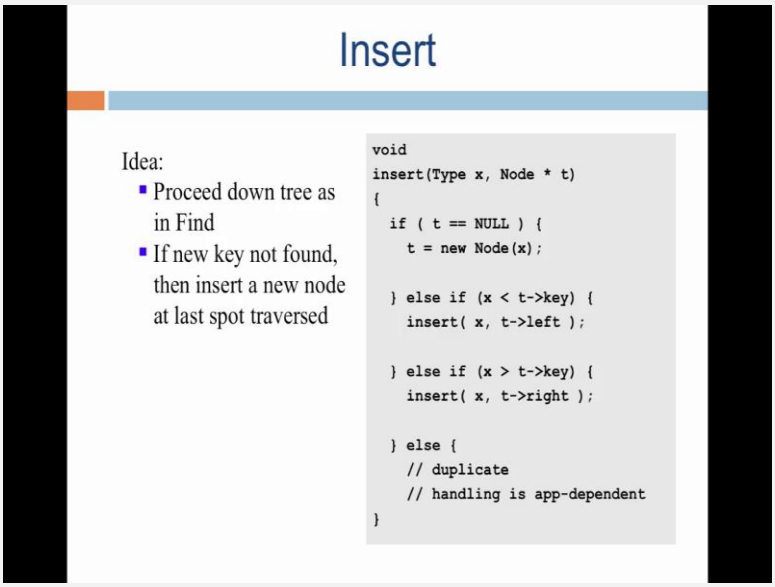
Iterate find is also conceptually fairly simple and this is one of those nice examples where the iterate code and the recursive code have more or less has same number of statements, typically recursive functions have much lesser statements than iterative implementations, but here is an example where the iterative implementations is more or less have the same length as the recursive implementation, actually it is even little smaller I would say.

Find key, ignore the comparable and from the pointer t, so all around I am saying is the key must come from a set which is a comparable set, this must have been written slightly differently, t is a pointer to the tree, what you do is you check, you explore the tree guided by the key. So, what is I say, if t is not null and t pointer at key is not equal to key, that is at the current node if you do not find the key that you are searching for.

If key is smaller you go down and search to the left, otherwise you go to the right and search for it at return t. Therefore, the iterative find kind of tells you that using the key value the search can be guided through the different links in the binary search tree, indeed there is a unique path as you can imagine, because of the order. For example, if I was searching for the value 6, let us I just check the iterative find will go first and 10 go to the left, because 6 is smaller than 10, here it will go to the right, because 6 is larger than 5, here the search will go to the left of 9, at 7 you will find that 6 is not equal to 7, then t then goes to the null pointer and then you return saying that null has been

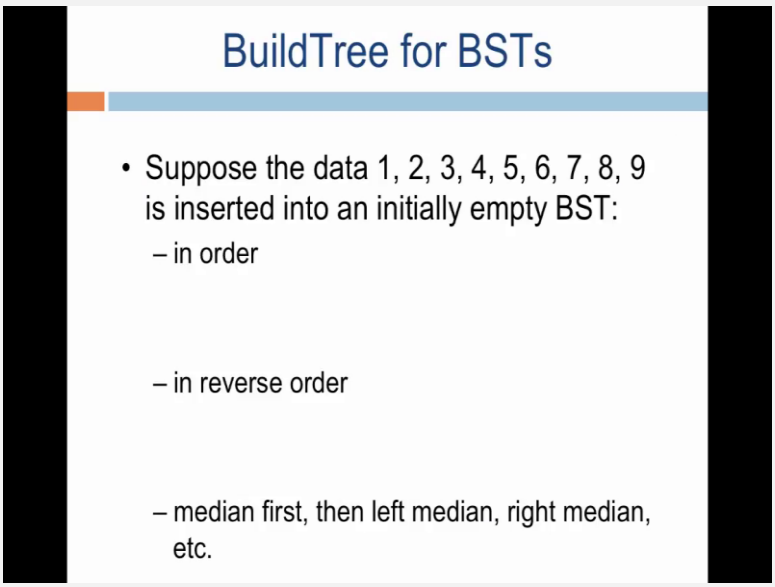
encountered that is a message that the key has not been found. Indeed you can use this to insert by doing this kind of search and inserting it whenever you find a null, we will see this again in a few slides.

(Refer Slide Time: 20:28)



Exactly, what I said proceed down the tree as in the iterative find implementation, when the new keys not found, ((Refer Time: 20:42)) let us just see this when the key is not found, then you will actually reach a null pointer, at that point you just insert it at that place, it depending on the order.

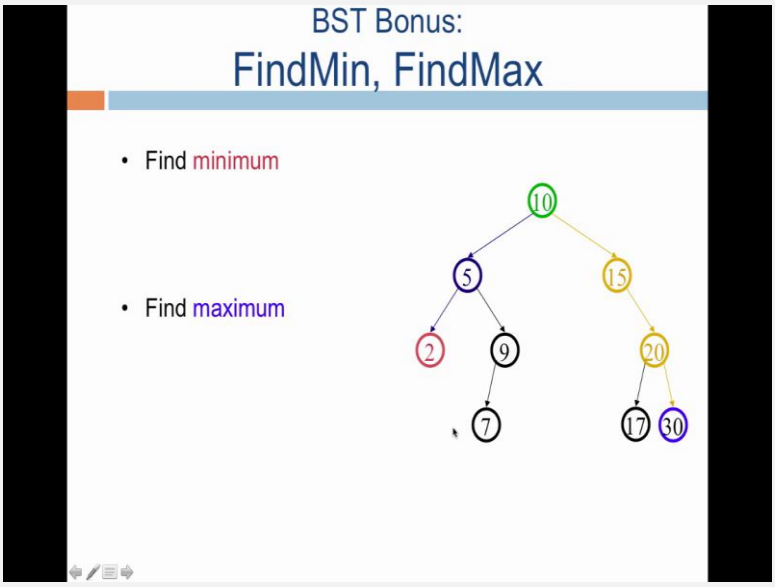
(Refer Slide Time: 20:56)



So, now let us just imagine that you have some data 1, 2, 3, 4, 5, 6, 7, 8, 9 inserted into a initially empty binary search tree and inserted it into this particular order or in the reverse order, let us assume that you have done this, now you can imagine here is a small exercise, you can check that is if you inserted in this order or in the reverse order, what is the depth of the binary search tree.

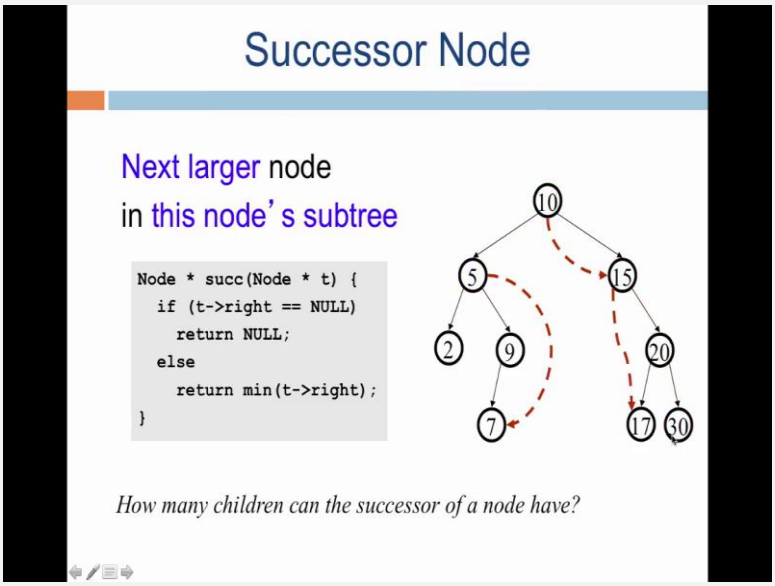
Similarly, if you inserted at the median first that is you inserted the middle element, let us say it is 5, the median element is 5 here, you inserted 5 first, then you have inserted the median to the left which should be 3, then the right median recursively if you did this, what would be the depth of the binary search tree. This is the comparison that you must perform. So, this is left as a small exercise for you to analyze and write down a formula for the depth of the binary search tree.

(Refer Slide Time: 21:51)



Here are two nice properties that I have mentioned in the few slides back, the smallest element of binary search tree is a left most leaf that is the first element in an inorder, first element to be printed in an inorder listing and the last element that we have listed in our order listing is the largest key in the binary search tree.

(Refer Slide Time: 22:26)



This observation is very useful for us to answer the following question which is called the successor node. So, let me ask you, if you look at this node the key here is 10, if you have to answer the question which is the node which contains the next largest value, which is the node that contains the value just larger than 10 in this binary search tree. As you can see the example is 17, but observe that the answer lies in the analysis in the previous question that the key smallest, if the binary search tree is the left most leaf and the key largest in a binary search tree is the right most leaf.

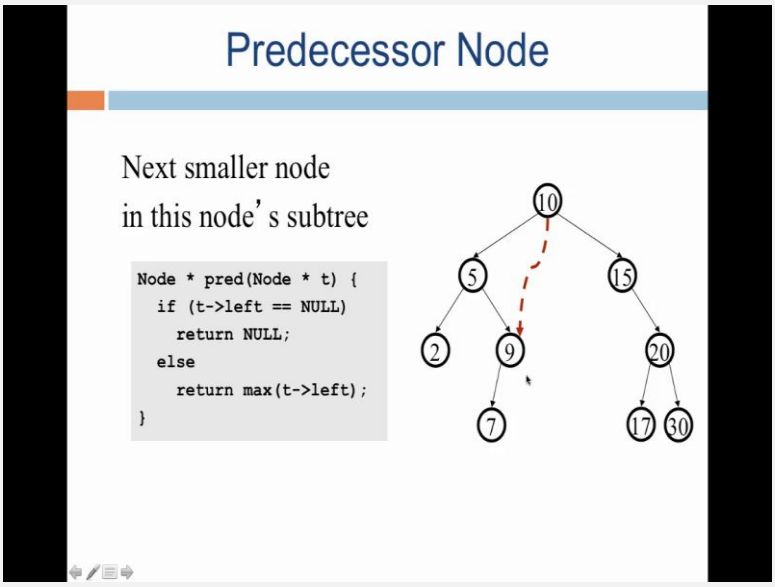
Therefore, the smallest key larger than 10 is here and the largest key smaller than 10 is here that is you go to the left child and pick the largest key and you go to the right child and pick the smallest key, let me repeat this again. Just in case I created confusion, you go to the left child and print the first value that is printed by an inorder listing. For example, the smallest key larger than 10 is 15 and observe that the inorder listing will print 15 first and that is this smallest value larger than 15, larger than 10.

The smallest value larger than 15 is 17 and it is the first key that is printed by the inorder listing, then you go to the right child. Similarly, the largest key smaller than 10 is 9 and if you go to the inorder listing 9 is the last value to be printed. Similarly, the smallest key larger than 5 is the value 7. So, this is the small piece of code if you want to know the next larger node in a certain node sub tree that is t is given as a pointer, so let us say it is

pointing here and you have to print out, return a pointer to the node that contains the next larger value.

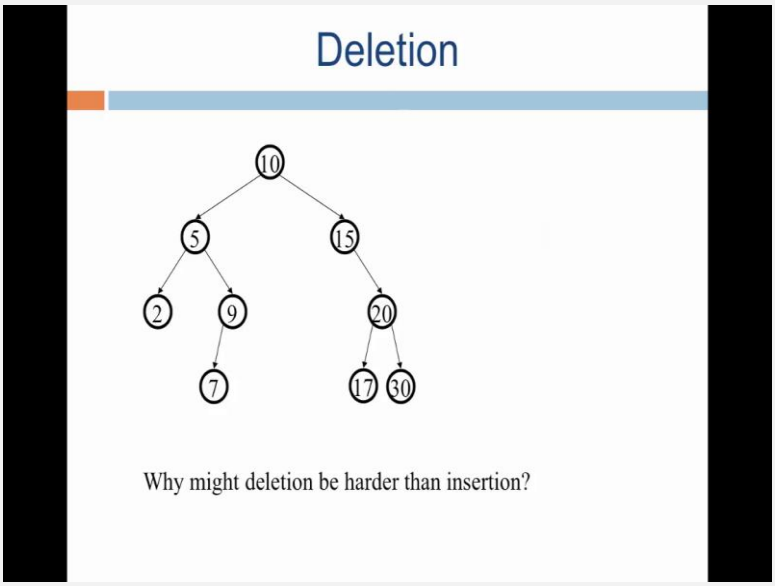
So, if the right pointer is null, because you are looking for the next largest value, then you say that the current node is a largest, in this sub tree there is no value which is larger, otherwise you return the smallest value in the right side. So, that is captured by this particular picture, I have three example nodes at 10, at 15 and at 5.

(Refer Slide Time: 25:36)



Same with the predecessor node, it is a symmetric you just check if the left child is null, if the left child is null, then you know that the key is a smallest value in the tree. Otherwise, you find the largest value in the left sub tree. For example, the largest value smaller than 10 is 9 and it is the largest value in the left sub tree, so this is the logic. As you can see, if you have done a binary tree implementation these are all very easy functions to implement apart from just the structural binary tree traversal, you only have to do these comparisons that is, the search properties you have to implement. Every ever it is just a very simple comparison to be done. I encourage you to write these pieces of code yourself by the code is already almost here in this slides.

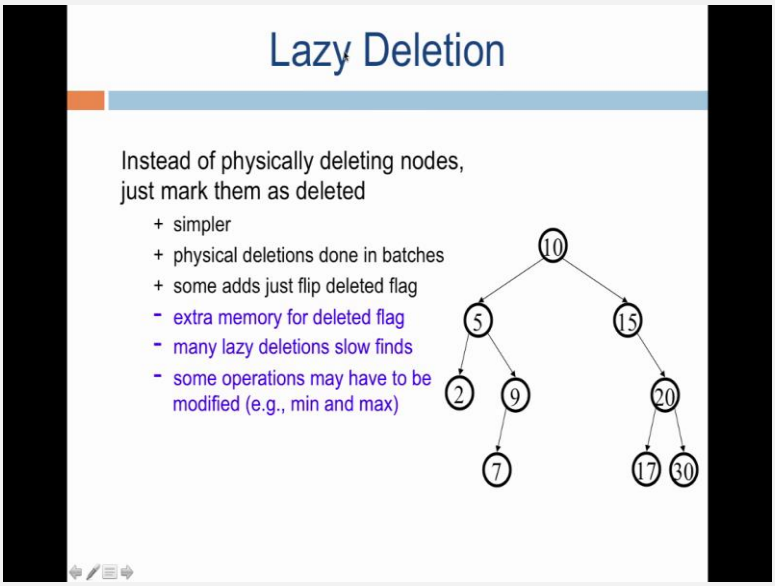
(Refer Slide Time: 26:28)



So, anyway come to the last topic when we come to talking about binary search trees, how does one delete a particular key? So, let us assume that would deletion be harder than insertion, insertion remember was quite easy, you just traverse the tree, guided by the particular key you want to insert, if you found that in to the tree you say that you already found it, if you did not find it you will reach a leaf node and inserted it as the appropriate child of that particular leaf node that is the insertion row.

Why is the deletions slightly more challenging than insertion, let us say you want to delete 5. How does one actually reorganize the binary search tree, so that you still have a binary search tree that is why deletion is conceptually little more challenging than insertion. So, let us do some three cases which will essentially illustrate the challenges of deletion.

(Refer Slide Time: 27:29)

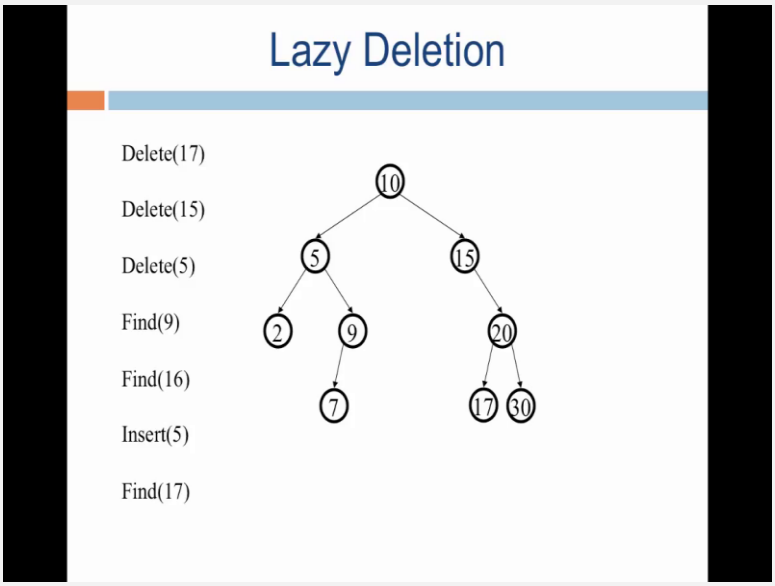


By the way deletion, can we done in multiple ways if you are writing programs, so you could just mark the node to be deleted, it is very quick and you can do the deletions steps once there are lot of deletions to be done. So, remember that if you are writing a program for maintaining a large binary search tree which is typically a dictionary of words and you are eliminating words from the dictionary, you might want to actually mark all the words and delete them in a batch.

Sometimes, something that you have wanted to delete if you wanted to re add it again, if you wanted to add it again, then you just have to flip the deleted flag. Therefore, these are the plus points, the minus points or the negative points is that you have to maintain a additional bit, additional flag for whether something has to be deleted or not. Because, you have this additional flag, if you want to do a find, you will have to check the value of the flag and you might have to change some of the pieces of code that you have already written for min and max.

So, if you have a binary search tree in this, some nodes are negative with delete flags and so on and so forth and then if you want to know what is the minimum we have to do some extra work. For example, if you want to know that what is the minimum in the binary search tree with 2 having the delete flag, then observe that 5 is the correct value and you will have to modify your min and max function to be able to get the correct answers. So, these are the challenges of deletion.

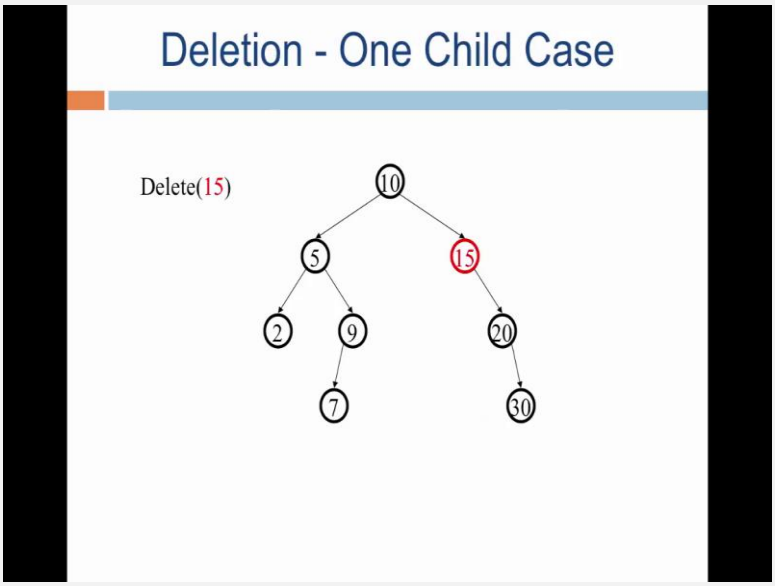
(Refer Slide Time: 29:00)



Let us do some lazy deletion in this example and each of these examples are interesting, you want to delete 17, observe that the 17 has no children. Deletion is conceptually easy, then you want to delete 15, now so this is the bit of the challenge, because 15 has a single child and we will see that conceptually whatever you will imagine would be right, you just take the right child and 15 does not have a left child and it has a only a right child.

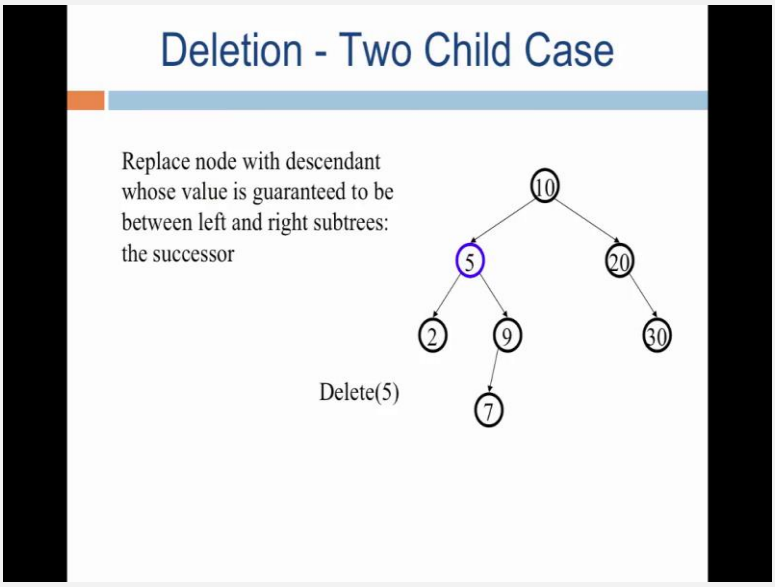
So, take the right child 10, make it the right child of the parent of 15, let me repeat this. You want to delete 15, 15 does not have a left child, you want to delete 15, delete it, through it out of the tree and take the right child of 15 and make it the right child of 10. So, is the well defined step, then when you come to delete 5, you have a challenge. If you delete 5, how does one make a resulting data structures binary search tree again. So, this is what we are going to look at, let us just see this.

(Refer Slide Time: 30:00)



Deleting 17 is easy just delete it, when there is only a single child it is again easy.

(Refer Slide Time: 30:10)

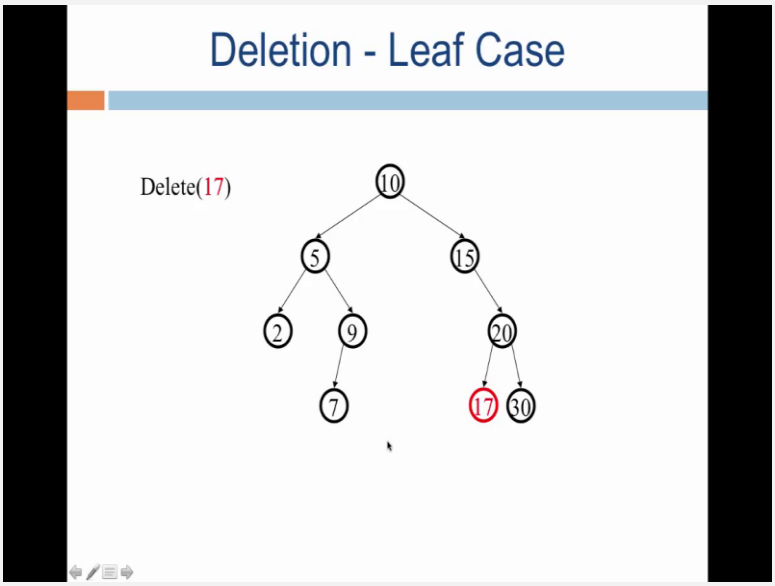


So, if you want to delete 15, just connect 20 to 10. Now, when you want to delete 5, what you do is replace some node with descendant whose value is guaranteed to be between the left and right sub tree, that is you pick a descendant which is the smallest value larger than 5 and copy it is value here, that is the simple delete. Let us go through the logic again. Whenever you come up with this challenging situation, all that you have to do is

go to the right child, pick the largest value. Observe that if it did not have a right child ((Refer Time: 30:47)) it is captured in this situation, in this case.

Now, it has both the children which means it has a right child. So, you go to the right child and pick the largest value in the right sub tree which is smaller than 5, pick this smallest value in the right sub tree which is larger than 5. In this case it is 7, you copy the value 7 here that is all, so somehow. So, this slides seems to have gone missing, so what you do is you take the smallest value larger than 5 by going to the right sub tree and take this value and copy it here and delete this node. So, this is the implementation the delete function, let me just go back.

(Refer Slide Time: 31:32)

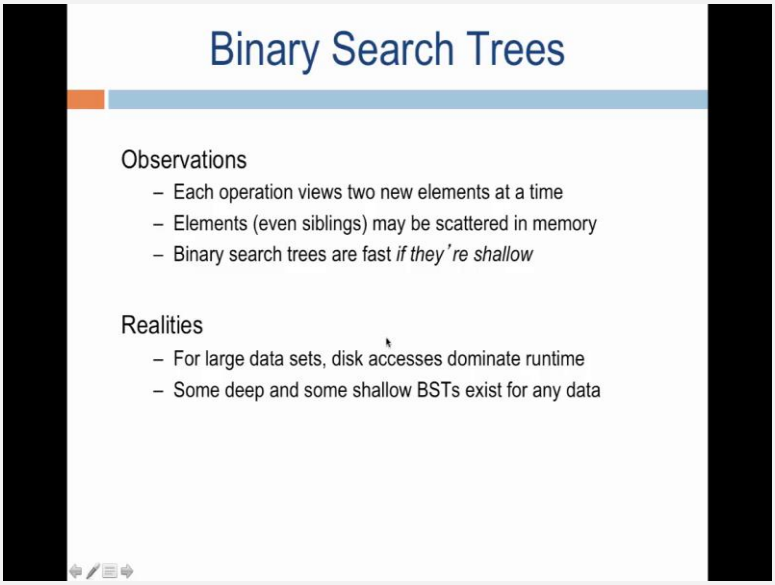


Observe that this is just a structural change, you go find 17, it is a leaf just delete the node from the tree, it is a structural change no work needs to be done. This also is a structural change after you have found the node that contains 15, you check that it has only one child and you take that, that is it has only a right sub tree, in this case you take the right sub tree and connected to be the right child of the parent of 15, that is clear.

The same is true if you have only a left sub tree at 15, may be you have 14 and 13 all that way you could do is take 14 and 13 connected to the right sub tree of 10, delete would be done. Now, let us come to the third case ((Refer Time: 32:19)), if you have two children then conceptually also it is not hard, but it is new. You want to delete 5, go to the right

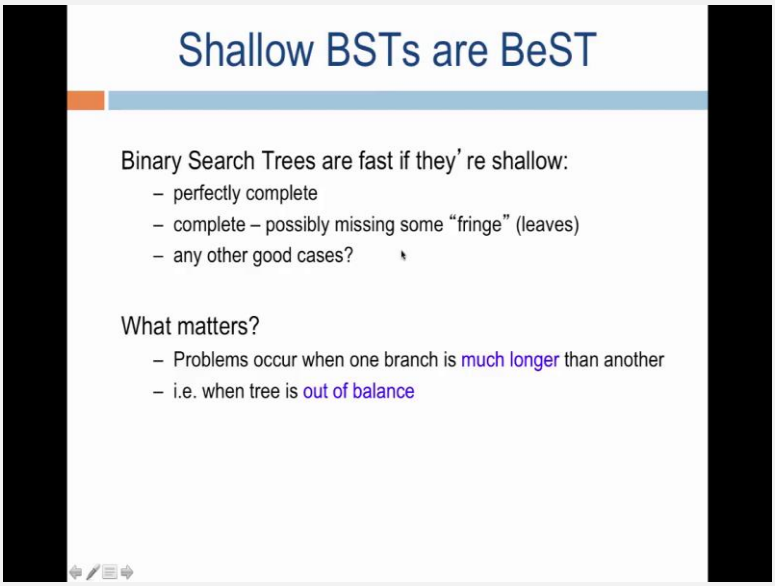
side, pick the smallest value larger than 5 and copy it here and delete that particular value from the tree, you have a binary search tree it is very easy to analyze.

(Refer Slide Time: 32:37)



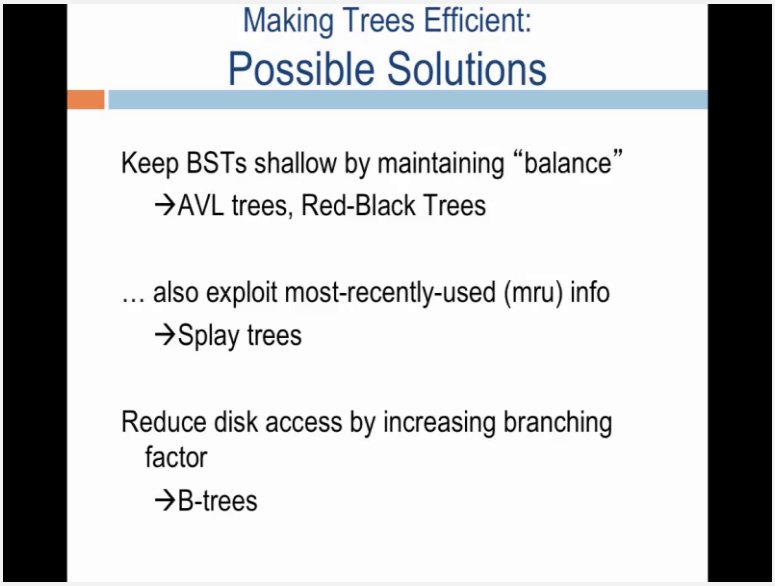
Therefore, what we have seen is that binary search trees are very nice, it is possible to insert, delete a query, conceptually they are very easy and they are very useful to implement dictionaries though I stored only values which are numeric values in the tree, it is possible to implement all these things for the other values too. So, other data types as long as they are comparable, that is why I said that you must have a well define comparison between any pair of keys. Now, when a binary search tree is the best, when they are shallow?

(Refer Slide Time: 33:15)



Shallow BSTs are best. So, I will allow this is an exercise for most of you to think about as to and this has already been given as a small exercise a few slides back, insert into a binary search tree in increasing order and in decreasing order and in the best order where the median elements are repeatedly inserted into the left tree and the right tree, respectively compare the depth of the binary search trees. The shallow at the binary search tree quicker the finds and quicker the update operations. So, therefore the challenge now is how does I maintain a shallow binary search tree or a low depth binary search tree.

(Refer Slide Time: 33:59)



There are multiple algorithms in the literature, in the text books, you can use balancing approaches for AVL trees, Red Black trees, you can also use Splay trees and then you can also use B trees. So, as you can see in this lecture, we have looked at the basic data structure, the binary search trees which are used for implementing dictionaries. All the methods as you saw are very simple modifications to the binary tree implementation that you have already done.

Now, the only things that you need to implement when you have to implement a binary search tree on top of a binary tree is that you have to ensure that the key values which are present at set at the nodes are appropriately combined and inserted. All the code was present in the slides and I encourage you to go out and implement them. In the next lecture we will look at the heap data structure.

Thank you very much.