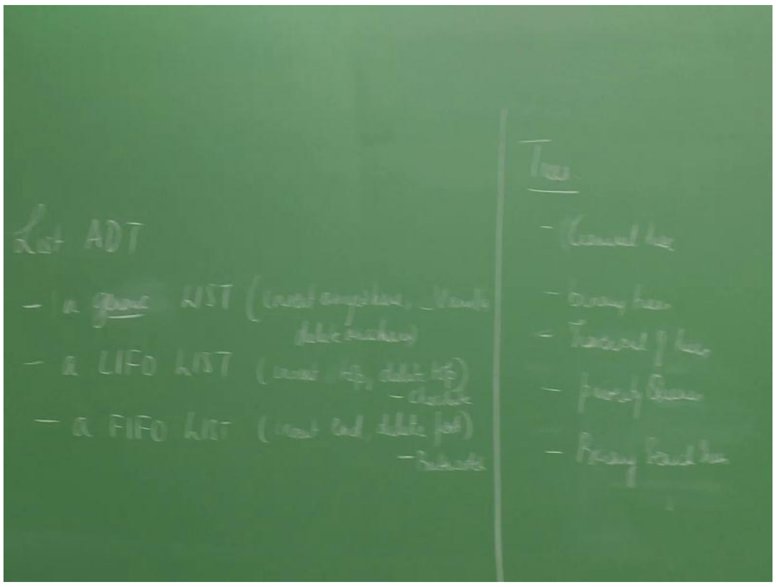
Programming, Data Structures and Algorithms
Prof. Hema Murthy
Department of Computer Science and Engineering
Indian Institute Technology, Madras

Module – 08
Lecture – 48
Summary of list, stack, queue
Introduction to Trees
Motivation using family tree
Representation of a tree: array representation,
linked list and leftmost child right sibling representation
Binary trees
Operations on binary trees: create, findchild, findparent

In the last class we looked at the list ADT.

(Refer Slide Time: 00:17)

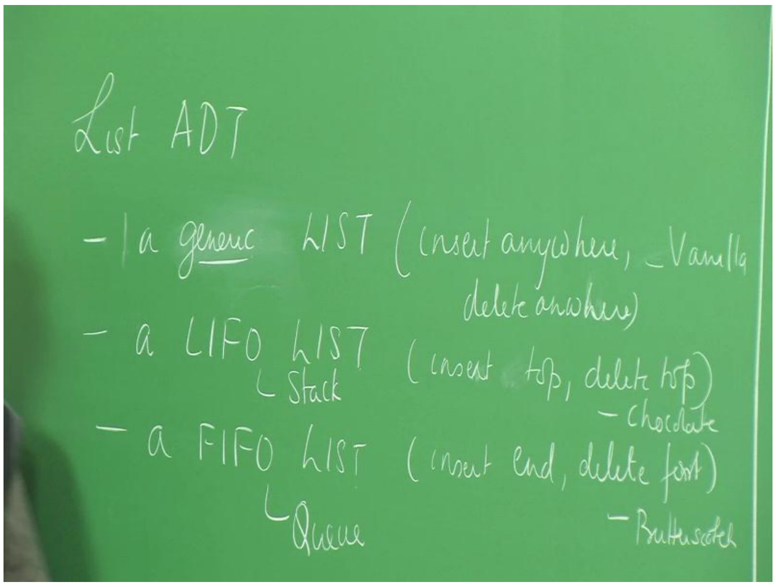


List ADT
- a generic LIST (insert anywhere, delete anywhere)
- a LIFO LIST (insert top, delete top)
  - [illegible]
- a FIFO LIST (insert end, delete front)
  - [illegible]

Trees
- General trees
- Binary trees
- Traversal of trees
- Priority Queues
- Binary Search trees

And we talked about a generic list ADT, there we said insert any element anywhere.

(Refer Slide Time: 00:30)



List ADT

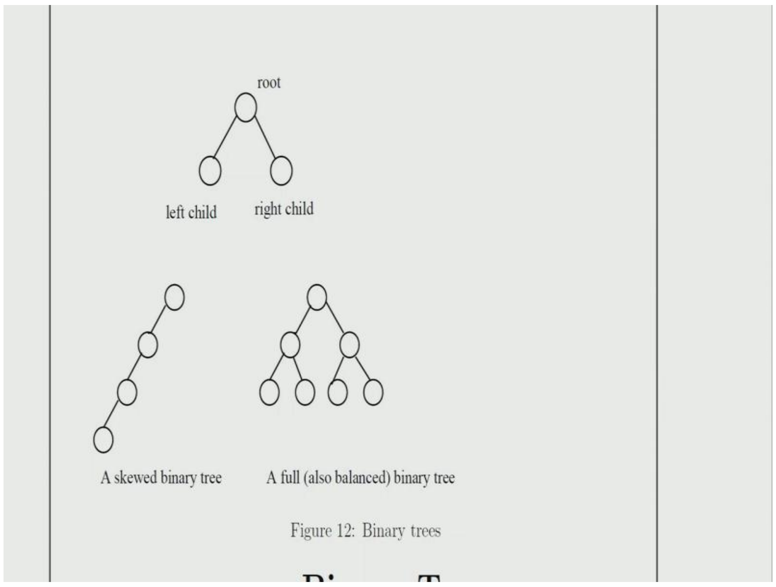
- a generic LIST (insert anywhere, delete anywhere) – Vanilla
- a LIFO LIST (insert top, delete top) – Chocolate
  └ Stack
- a FIFO LIST (insert end, delete front) – Butterscotch
  └ Queue

And then we have talked about two other flavors of the list ADT, one is the stack LIFO Last In First Out and the other is the FIFO which is the First In First Out or the queue ADT. Now, the generic list ADT what it consists of, it consisted of a sequence of items. So, it had a set of elements and if you want you could have completely an unordered set of items as in the generic list or in the last in first out of the stack ADT, we had a particular flavor where you said you can only insert at the top and delete from the top.

And in the first in first out this is the queue ADT, we inserted it at the end and we deleted from the first. So, it is like you know when you look at generic ADT, it is like a vanilla ice cream when you know, whereas the stack ADT is like chocolate and if you like chocolate, then it is like chocolate ADT which is does something very special and on the list ADT. And a first in first out is perhaps like butter scotch which just something else special.

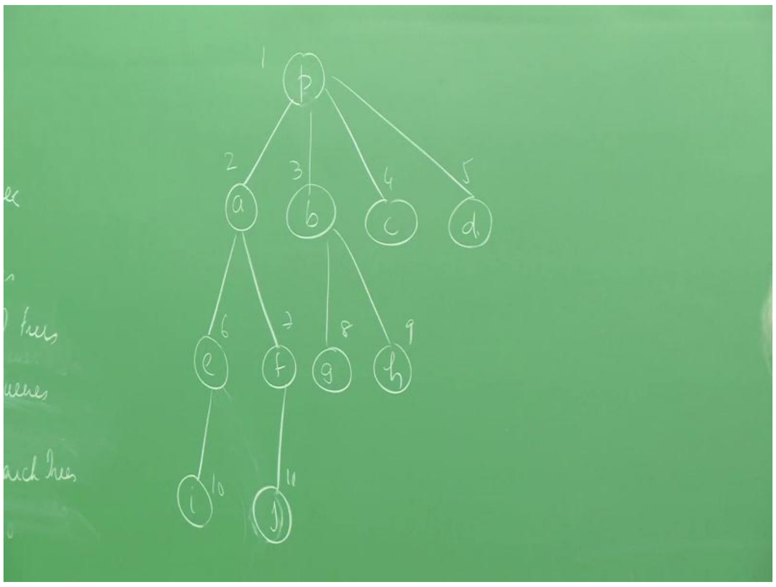
So, it depending upon what you like, I do know these are some of the most favoured flavors in ice creams. So, it is something like that, so either a stack or a queue still qualifies as a generic list now, there is one problem with the stack or a queue and the problem is the following.

(Refer Slide Time: 01:49)



What we see is that whether directly or indirectly there is some kind of ordering in the list. The items either or pushed on to the stack and popped from the stack in one end or they are, you know pushed at one end, removed from the other end as in the queue ADT. Now, today what I want to talk about is, let us say I want to represent the family tree.

(Refer Slide Time: 02:25)



Suppose I want to represent my family tree, let us say a is the parent has b e, a set of children let me say that p are the parents it does not matter whether mother or father. And let us see you have four children and these are your siblings, now this let us clearly the old generation and this generation now for example, generally has generation 1 above you guys has had perhaps only 2 children f and so on.

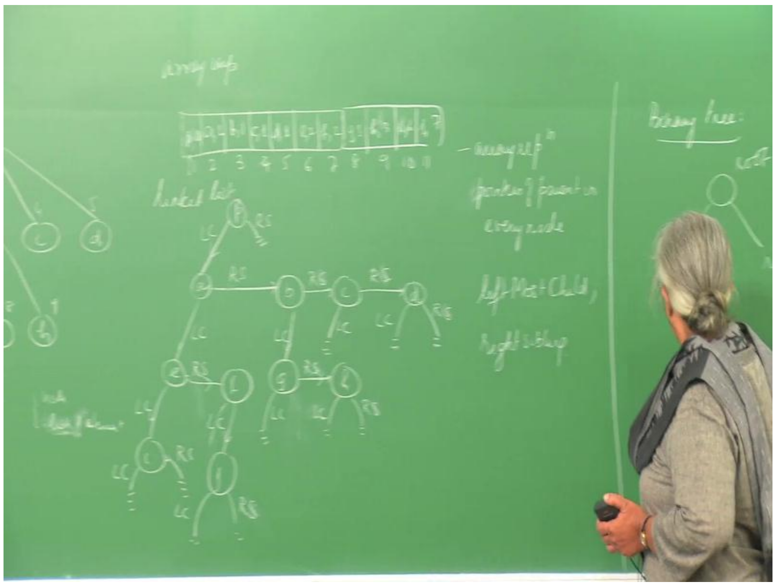
And suppose you want to find out, you know who is the grand parent of whom and so on, then if you want to traverse you have to represented all of this using a list, then the problem would have been, this would have been number 1 in your list 2, 3, 4, 5, 6, 7, 8, 9. Assuming that 6 came after 5, you put them in this particular list and to access an element at 10, let us say or to access the element the 11th element for that matter, you do have to go to the end of the list.

So, this would have, but at the same time the reason I am talking about this is when I am using a list to represent something like this, then what kind of a list will I use, I will use a generic list. Because, there is nothing like when you look at siblings for example, there is nothing like a priority of one sibling over the other. But, just the representation by itself it is, first you put them in and so on and so forth. But, you know deletion for example, someone dies in the family, it can happen at any point in the list and all most all operations will become very expensive, because you have to traverse mostly the entire list. So, the idea is can we represent these kind of structures in a better way. So, one of the most popular representations, what do we want from this, if I want to access the node corresponding to j, then I would like some information.

What would like I to say I want to look at, then I would like j to say who is his or her parent, who is this one’s parent, who is this is parent if I want to find a ((Refer Time: 05:20)) of j. So, how easy or how difficult is this depends upon the kind of representation that we are going to use. So, a tree data structure kind of suites this kind of data very well, so what we are going to talk about is we are going to talk about a general tree something like this.

And then as I already said, the major objective is to be able to traverse this particular tree, see if a particular element is present in the tree based on certain criteria that can be satisfied and so on. We will come back after we do all these to generate a tree, but this is in general a tree, now representation of the tree can be done in a number of ways.

(Refer Slide Time: 06:16)



You can use an array representation or you can use what is called any general tree is represented by what is called the left most child, right sibling representation. What this means is p, a, b, c, d is the representation for this and this is represented like this and you have i which is represented like this, it has two children f and this one has the left most child j, I will explain in a minute what this representation is all about.

So, what is done is notice that we take this particular tree over here and converted it into to this format what would I do if I was representing an array, I would say here 0 this is p, let us say this node corresponds to p. Now, I keeps some information 0 over here, then the indices go from 1 to 2, I would say a comma 1, b comma 1, c comma 1, d comma 1 and I would say e comma 2, f comma 2, j comma 3, I will tell you in a minute what I am representing over here, h comma 3 and i comma 6 and j comma 7. What I am storing here, if I am this is an array implementation and this is a link list implementation.

What are the link list now, it is a recursive data structure, what is the problem with this tree, we want to represent this tree, then using a link list then at every node I must be able to say, if I was representing it by the number of pointers I must say that node has some max number of pointers n. To avoid that you know, I could have a list of elements on the other hand that is exactly what we are doing over here.

See, you looked at it over here after that we are doing, we are saying here p is a root node and it has children a, b, c, d. Let us say a is the left most child or the oldest child, then to make our representation more convenient, if this is the root node within that it has a pointer to the left most child which whose right sibling point to all the other nodes.

And similarly it will also have a pointer to it is left most child, so the tree is like this, the pointers that are pointing forward. On the other hand in the array implementation, why are we doing is, because as I say one of the fundamental problems says we want parent information or from the parent I would like to go to the child, one of these two information is required. See if we look at this, the array implementation on the other hand what do we do, we say that I store the parent in this particular, let me call it the index of the node is 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 and 11 suppose I have storing it in an array.

Then, what we do is we store a pointer to the nodes parent and now I have put the index 0, index 0 means what, this particular node is the root node. Next what I do is the element a, now element a is stored here and who is it is parent, it is parent is p whose indexes is 1. Similarly, the node b is stored in index 3 and it is parent a is a node 1 and I keep that index. Now, for c what happens a, b, c, d is like that, when I come to e what do I do, I keep the index of it is parent which is two, because a is the parent of e therefore, I store e and 2 over here.

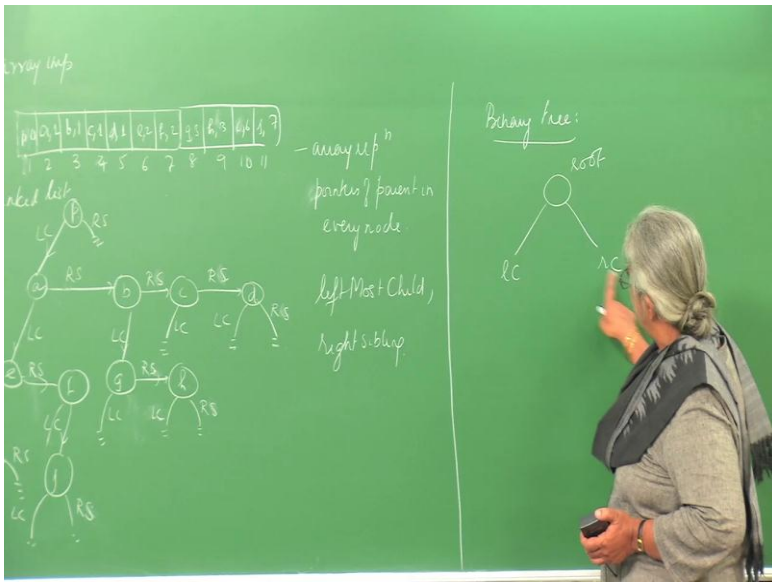
Similarly, for f what do I do, I store f and 2 over here, because for f the parent is again a. So, you repeat like this and you can get use the array representation for any general trick. In the link list representation, this representation is basically this is array representation where we use pointers to the parent in every node. On the other hand, in the link list representation what does every node have, every node has a pointer to it is left most child and the right sibling.

So, notice that p is the parent, it is the root of this particular tree therefore, it has only pointer to it is left most child, it is right sibling has pointer to null, because the root has no siblings. And here what is that is happening here, a is the left most child and it is pointing to the, it is right sibling this is right sibling, this is it is left most child, left most child, left most child, right sibling pointer, left most child, right sibling, right sibling, left most child, left most child, right sibling, left most child, right sibling, right sibling, left most child, left most child, right sibling.

So, what I have done now in all the arrows now I have put the which corresponds to the pointer, which corresponds to that of the left most child and right sibling, RS to the left most child and right sibling. So, we make these pointers like this. Now, that means what is nice about the structure, in every node we are having only two pointers the left most child and a right sibling.

But, so basically you have two pointers in every node, but you are able to represent any general trick. Now, this tree is useful if you want to look at any arbitrary tree. For the time being we will kindly postpone this and we will talk about this is a little later. But, remember that this is one such representation of a tree. What is the meaning of the left most child, left most child is older sibling and a right oldest child, in the right sibling corresponds to it is next sibling at that particular 11. We will come back to this history a little later where we have either array representation of this.

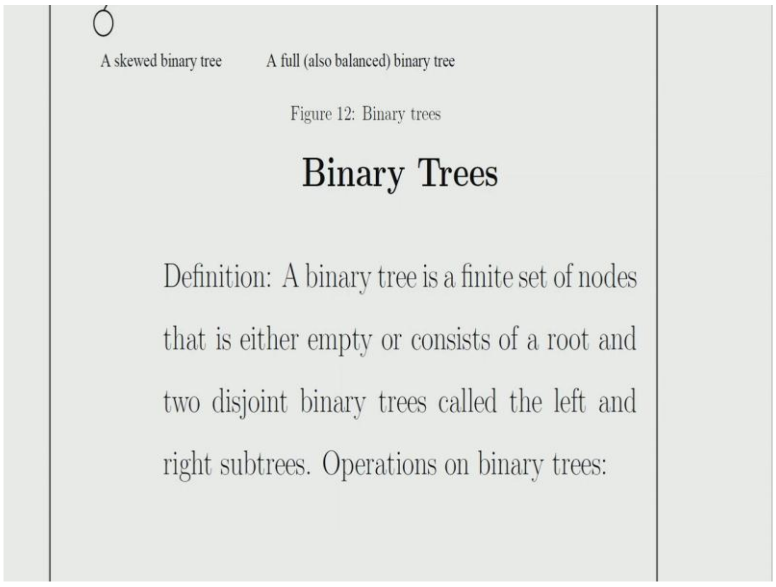
(Refer Slide Time: 14:36)



What we are look at is we look at some special trees today. In particular we will look at the binary tree. What is the binary tree now? Here is an example, where we have the root and it also has two pointers one to the left child and the right side, you know hierarchy here, that is the fundamental difference. It is similar to this where you have a node has two pointers left most child and right sibling. The difference between the binary tree node and the general tree node is that the left child and right child there is no hierarchy.

What you mean by that here a points to b, two had been nodded at the same level whereas, the left most child points to a node at the one level lower. So, basically what we are talking about here is one level higher, here in the other hand you are saying left more child and right child, left child and right child are at the same level. Now, what is the use of such trees, we will see in a minute.

(Refer Slide Time: 15:43)



A skewed binary tree A full (also balanced) binary tree

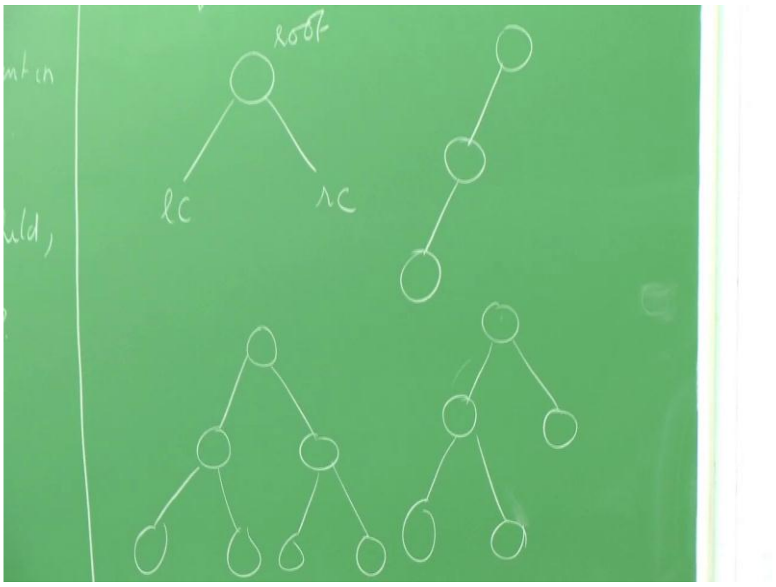
Figure 12: Binary trees

Binary Trees

Definition: A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left and right subtrees. Operations on binary trees:

Now, how do we define a binary tree, it defined like this, what is it. It is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left and right sub trees.

(Refer Slide Time: 16:02)

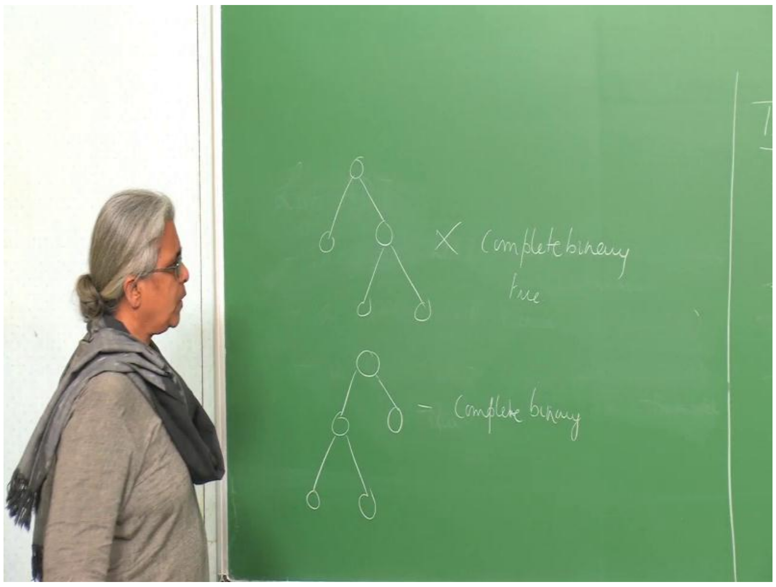


What is it telling me that, this is the binary tree, it can also be empty and it has left and right sub trees which are again binary trees. Again going back to the example, if you started from this node, this is the tree still and this is also a tree, this is also tree it has only one node. But, basically what we are saying is at every level the same structure follows again from every node, left most child right sibling, it is what we said.

Similarly, in the binary tree here also every node can have optionally two sub trees, it can also have binary sub trees or it need not have any sub tree at all. If it does not have the root, even an empty tree is a binary tree. So, this is a skewed binary tree, the picture is given here in the slide, skewed means, it is skewed to the left and then I have drawn other binary tree which is called a full binary tree, the nodes are full like this.

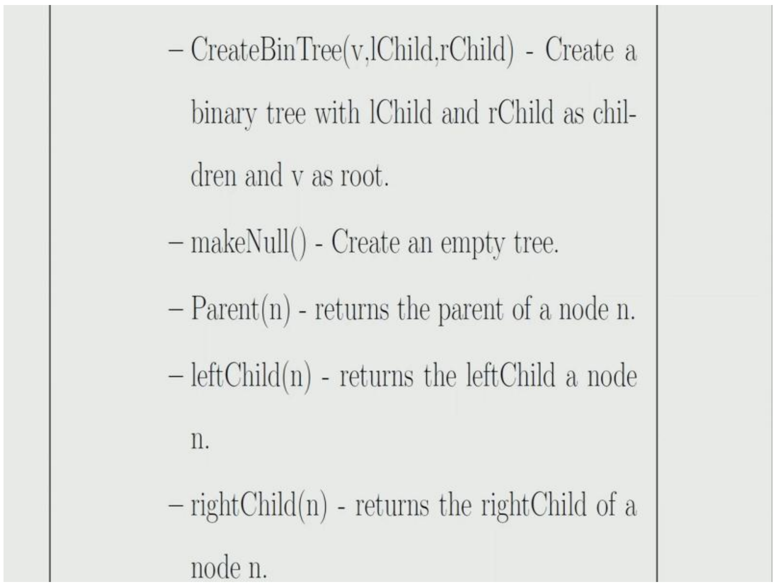
Then, you also have another variant of it called the complete binary tree, we will come back to this. What we mean by this is when you fill this tree for example, notice that I filled the nodes with the first level, nodes with the second level, nodes with the third level when I am filling them, I am go from left to right. This is the complete binary tree.

(Refer Slide Time: 17:45)



While this one is not, this is not a complete binary tree, while it is a mirror image this is a complete binary tree. Such binary trees have a lot of applications in building what are called priority queues we will talk about that a little later. First let us see, why do we want a binary tree in the first place and is there any use of this and what are the operations that you will perform on a binary tree.

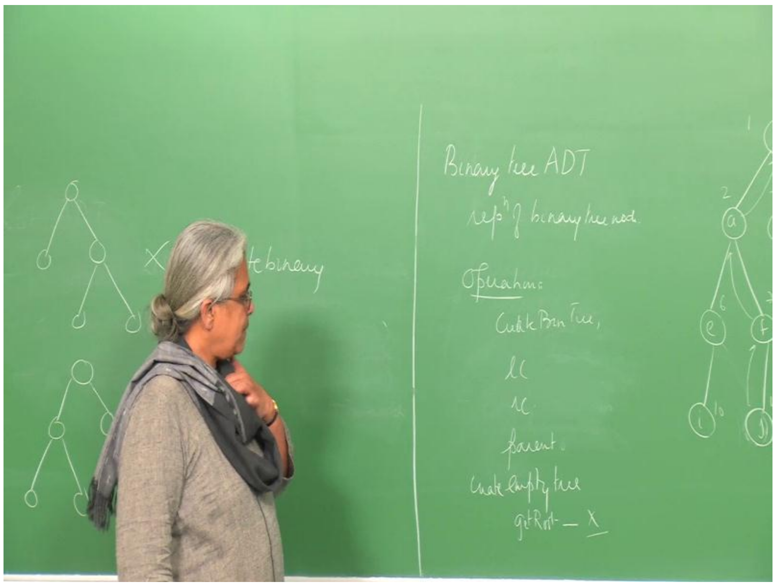
(Refer Slide Time: 18:34)



– CreateBinTree(v,lChild,rChild) - Create a binary tree with lChild and rChild as children and v as root.
– makeNull() - Create an empty tree.
– Parent(n) - returns the parent of a node n.
– leftChild(n) - returns the leftChild a node n.
– rightChild(n) - returns the rightChild of a node n.

So, you need something, so what is this that we do, these are the... So, now what is the binary tree now? The binary tree is also an abstract data type, let we saw the stacks, queues and list or list, queues and stack and see you have up…

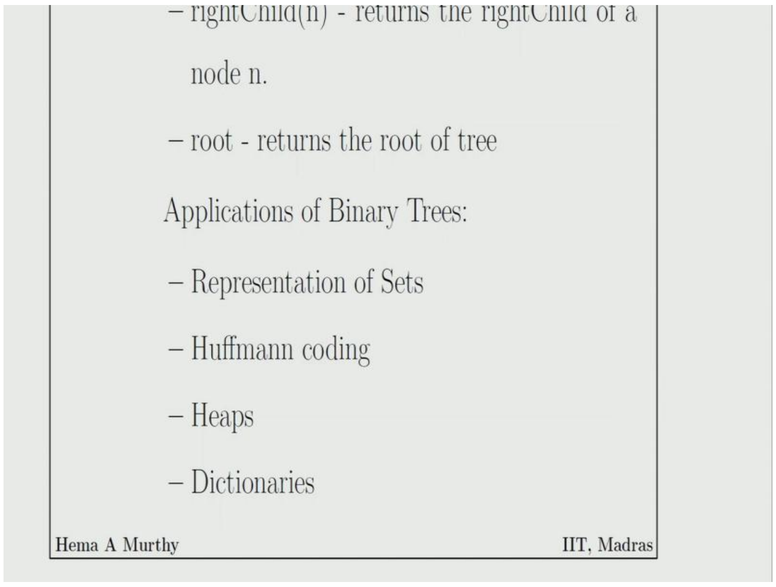
(Refer Slide Time: 18:56)



So, we have the binary tree ADT and abstract data type that we have a representation, the Huffman ((Refer Time: 19:06)) representation says there is a root node and the left child and a right child that is the representation binary tree node. So, essentially a recursive data structure and then we have operations which can be performed and what are the operations that you performed, because an ADT is like a mathematical module and it perform various operations on it.

So, what you want to do, you want to create a binary tree, first create an empty binary tree or create a binary tree with a particular node. Because, you have to start from somewhere, then you want to find left child, right child or right sub tree, then you want to find a parent of a given node and right child, left child or tree talked about.

(Refer Slide Time: 20:20)



– rightChild(n) - returns the rightChild of a node n.

– root - returns the root of tree

Applications of Binary Trees:

– Representation of Sets

– Huffmann coding

– Heaps

– Dictionaries

Hema A Murthy IIT, Madras

And of course, we need to create empty tree always that should required, parent and not always, sometimes they ask for the root of the tree. So, creation of empty tree and sometimes get root, this is not good to have, but since some other books talked about it, I kept this particular operation, because get root if you have it tells you what the implementation of the binary trees. So, now what we will do is, we will take one application, there is a lots of applications of binary trees, you can represent sets, you can do Huffman coding, you can represents heaps and dictionaries.