Programming and Data Structures

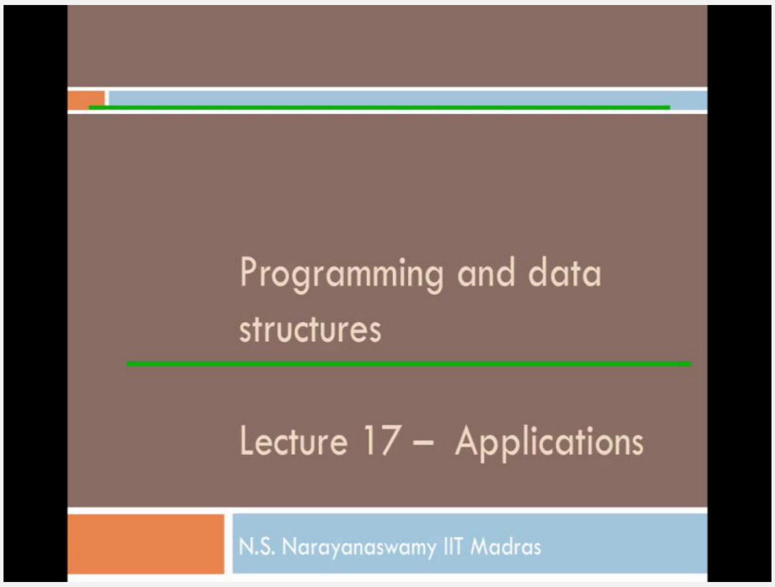
Prof. N. S. Narayanaswamy
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture – 16
Tree Applications

Welcome to lecture 3 on the tree abstract data type, and in today's lecture we are going to look at a whole sequence of formulae that one can write down on a binary tree. And each of these formulae are very important and these are formulae for which you should write functions using the binary tree ADT, and the tree ADT that we have defined and implemented in the previous lectures.

In today's lecture you will not see any C program that I have written, but a complete analysis of the different formulae that we want to write associated with binary trees. And you will also see some of the examples that were hard coded into the C programs in today's lecture in the slides. So, today essentially going to the presentation based on slides and there will not be any C program.

(Refer Slide Time: 01:11)



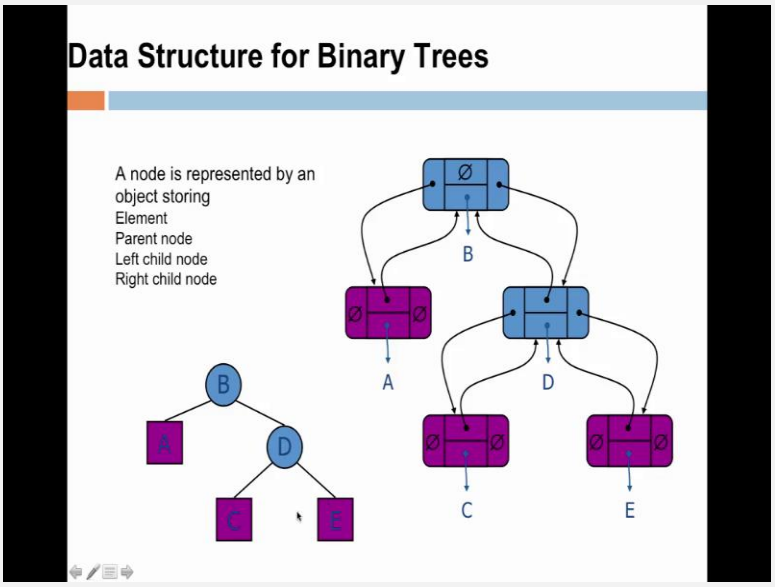
Programming and data structures

Lecture 17 – Applications

N.S. Narayanaswamy IIT Madras

Let us go to the slides and move on with a study of Applications using the tree ADT.

(Refer Slide Time: 01:18)



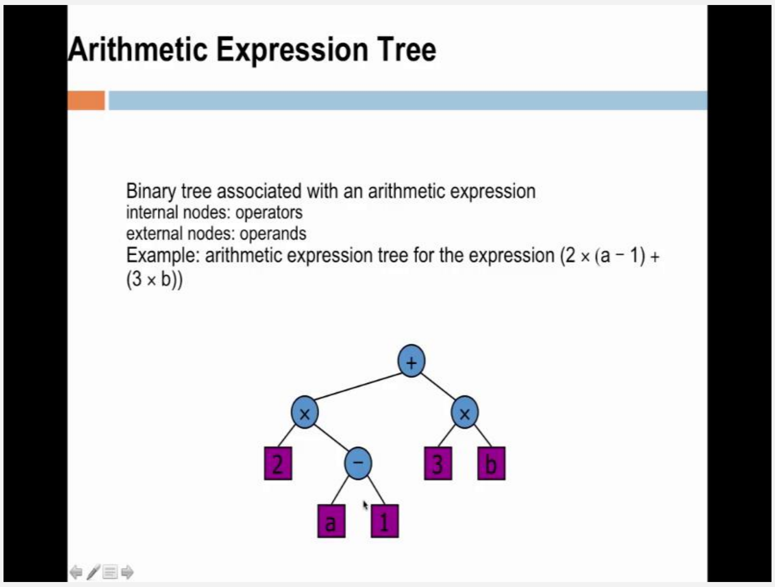
Data Structure for Binary Trees

A node is represented by an object storing
- Element
- Parent node
- Left child node
- Right child node

In particular, we will be using binary trees and let us look at the in memory visualization of a tree. This is a node, it has three pointer fields, the left pointer, pointer to the left child, the pointer to the right child, the pointer to the parent, and the field that contains the data value.

And this is the abstract view of this picture that this is the node that contains B, the left child of this node is the node that contains A which is the leaf node, and the right child of the node that contains B is the node that contains D which is not a leaf node, it is an internal node and this as you can see is a binary tree in which every node has two children. Every internal node has two children and every external node by definition has zero children.

(Refer Slide Time: 02:11)

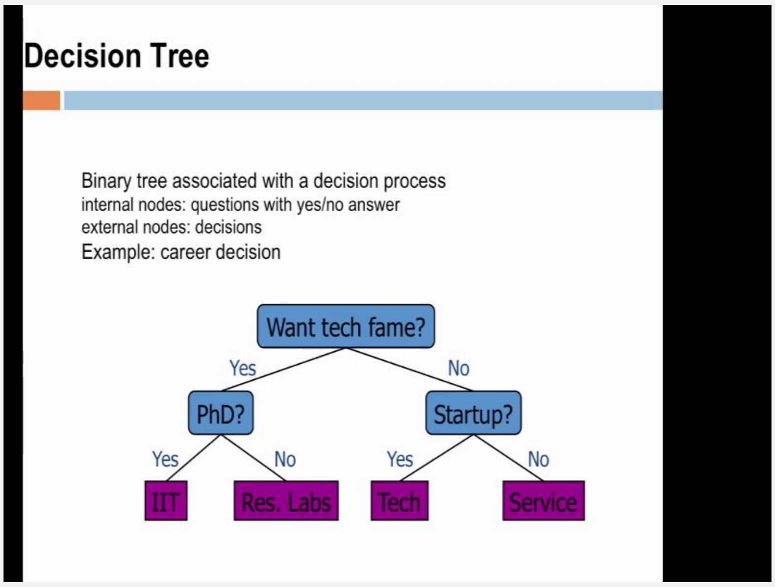


Arithmetic Expression Tree

Binary tree associated with an arithmetic expression
internal nodes: operators
external nodes: operands
Example: arithmetic expression tree for the expression (2 × (a − 1) + (3 × b))

Now, let us look at in an arithmetic expression tree. So, therefore observe this tree, this binary tree is used to store an arithmetic expression. So, now most of us has familiar with this way of writing arithmetic expression, but inbuilt into this way of writing the arithmetic expression and the way in which we evaluated is actually a binary tree. So, observe that this binary tree encodes this arithmetic expression.

(Refer Slide Time: 03:00)



Decision Tree

Binary tree associated with a decision process
internal nodes: questions with yes/no answer
external nodes: decisions
Example: career decision

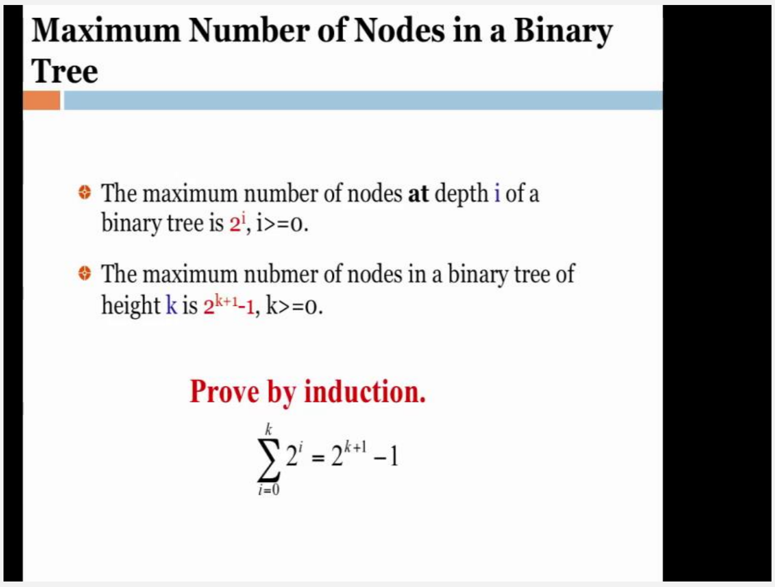
Want tech fame?
Yes
No
PhD?
Startup?
Yes
No
Yes
No
IIT
Res. Labs
Tech
Service

So, let us look at another application of the binary tree, you can think of this as a data structure which can store decision trees. So, what is a decision tree? This is essentially a tree that helps you get answers to a sequence of questions. So, let us consider a sequence of questions do you want to become famous as a technologist? If your answer is... If you, do you want a PhD, if your answer to that question is a yes, then you can register for a PhD at an IIT or at one of the research labs and if you do not want the PhD, then you want a startup.

So, if you want the technology fame, then if you want the PhD, thus the answer is yes to the question, then you can either do a PhD at an IIT and if the answer to that is no, then you would want to do a PhD at the research lab, and if you do not want a phD, but you want tech fame then maybe you want a startup and if you want to startup, do you want to tech startup and if it is, yes to a tech startup, you can go and create a tech startup and if it is not a tech startup, do you want a service startup.

So, this is an example in career decision, so therefore binary trees can be used to encode decisions that are made, I mean, so one can modify this image and one can put in all kinds of other decisions that you would want to take.

(Refer Slide Time: 04:46)



Maximum Number of Nodes in a Binary Tree

- The maximum number of nodes **at** depth i of a binary tree is 2^i, i>=0.
- The maximum nubmer of nodes in a binary tree of height k is $2^{k+1}-1$, k>=0.

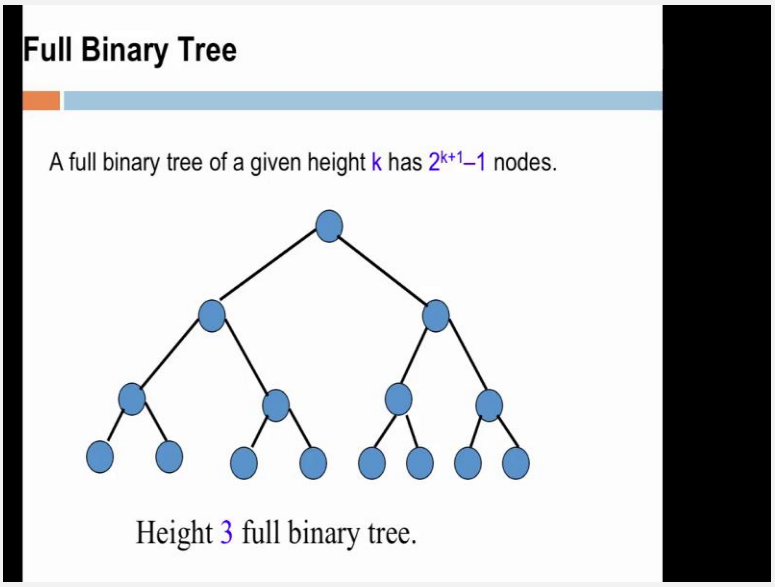
Prove by induction.

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

So, let us look at some parameters associated with a binary tree. So, for example how many nodes, what is the maximum number of nodes which are possible in a binary tree? The answer is very simple, it is the maximum number of nodes in a binary tree of height k is obtained by adding up the total number of nodes which are available at each level i. So, at a depth i the total number of nodes that are possible is 2 to the power of i for i greater than or equal to 0.

That is at the root node there is only one node, at depth 1 there can be at most two nodes, at depth 2 there can be at most four nodes, and if you add up the whole lot of them, then you essentially get 2 power k plus 1 minus 1. This is really does not require any prove by induction though I have written a prove by induction, you can definitely write a prove by induction if you want.

(Refer Slide Time: 05:50)



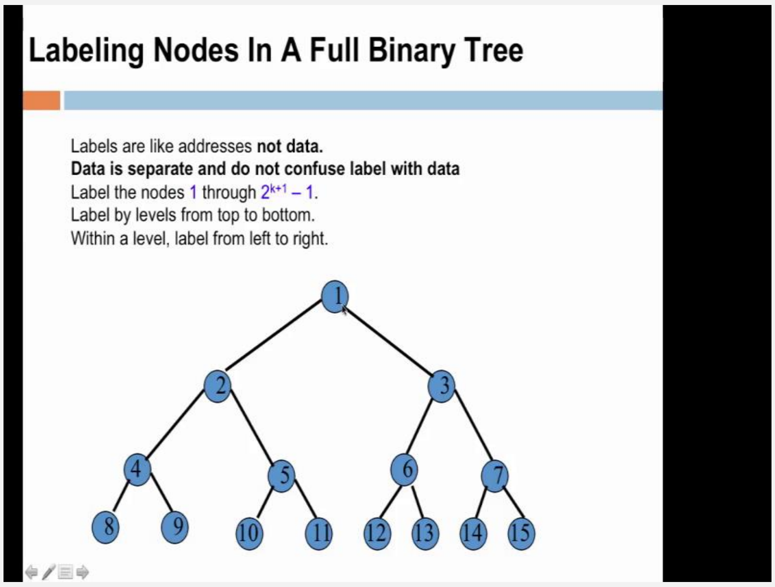
Full Binary Tree

A full binary tree of a given height k has $2^{k+1}-1$ nodes.

Height 3 full binary tree.

How many nodes are there in a full binary tree? A full binary tree is a tree which has all it is nodes and as we have seen in the previous slide, this is 2 power k plus 1 minus 1, where k is the height of the tree that is this is the depth 0, this is the depth 1, depth 2 and depth 3. This is the binary tree, a full binary tree of height 3 and you can actually verify that the total number of nodes which are there are 15 of them. So, you will see 8 here plus 4, 12 here plus 3, 15 nodes.

(Refer Slide Time: 06:26)

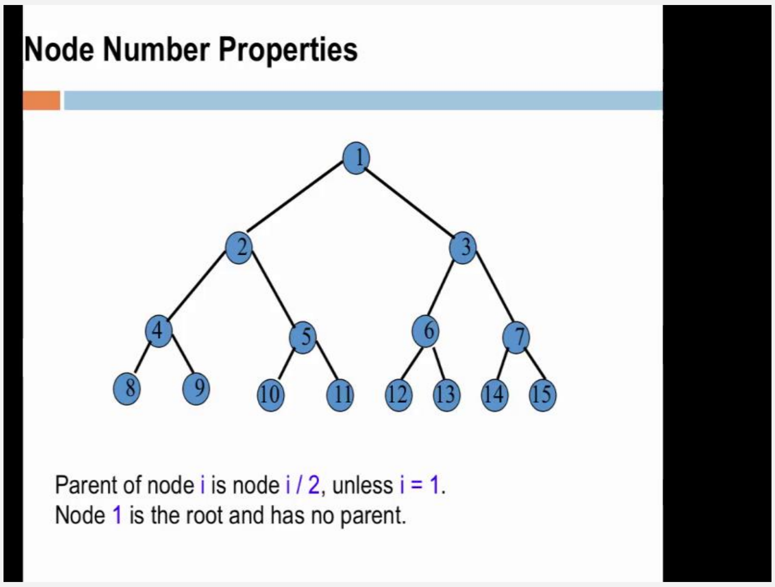

Labeling Nodes In A Full Binary Tree

Labels are like addresses **not data.**
Data is separate and do not confuse label with data
Label the nodes 1 through $2^{k+1} - 1$.
Label by levels from top to bottom.
Within a level, label from left to right.

How does one label the nodes in the full binary tree and this is extremely important and in this slide I would encourage the listeners or the readers to actually not confuse the data item which is typically stored inside a node with the label. In this case, this node has label 1, this has label 2, label 3 and so on and the way we do it is to use increasing order of labels as we go from top to bottom that is from the depth 0 onwards to the maximum depth and inside a layer we use increasing order of distinct labels from left to right.

So, now observe that the order is very important. For example, the left child of 1 is 2, left child of this node is a node label 2 and the left child of the right child of this node labelled 1 is the node label 3 and so on and so forth. So, therefore you can see that the labels are 1, 2, 3, 4, 5 and the interesting thing is that if you did a preorder traversal... Let we, you have some very special properties if you did a preorder traversal, let us look at what the special properties are.

(Refer Slide Time: 07:57)

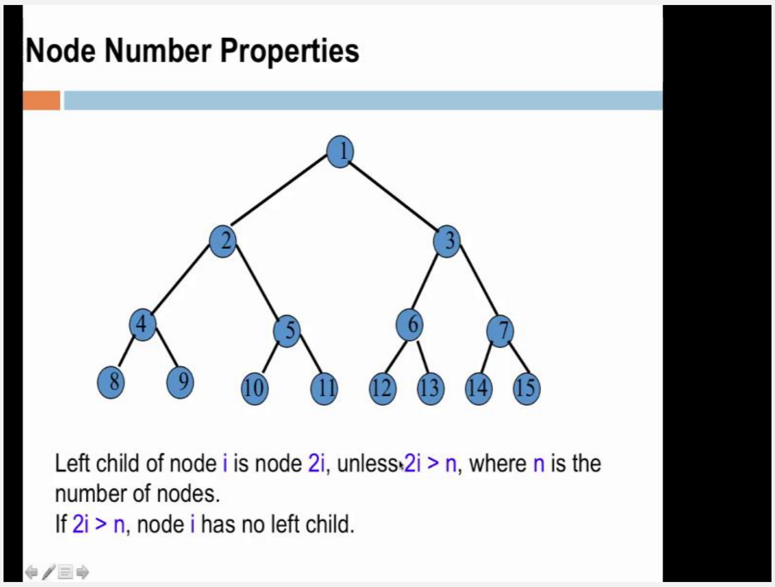


Node Number Properties

Parent of node i is node i / 2, unless i = 1.
Node 1 is the root and has no parent.

So, now let us look at the properties of the node numbering scheme. So, for example the parent of a node i gets the label i divided by 2 unless i is equal to 1 and there must have been a sealing here, this is the integer division. So, I take the course to C programming language, so the parent of the node 2 is one parent of the node 3 is 3 by 2. This is the integer division in C and that is also a value 1, same as true with every node.

(Refer Slide Time: 08:34)

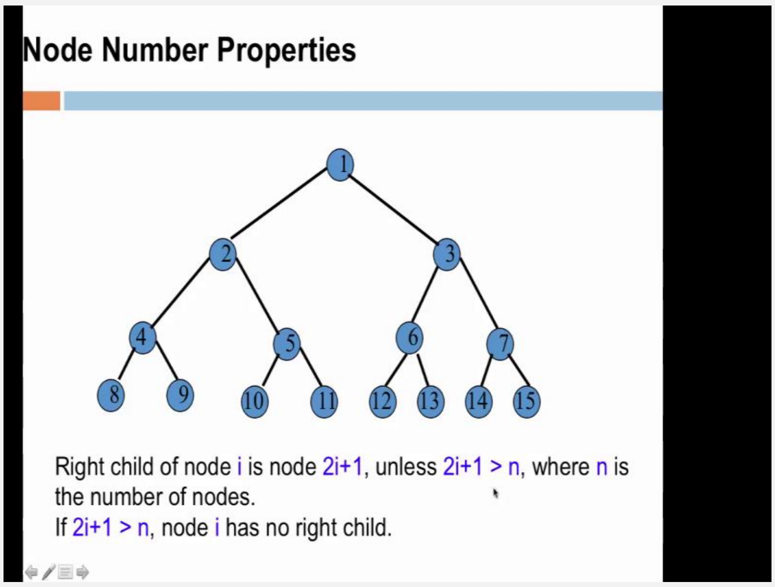


Node Number Properties

Left child of node i is node 2i, unless 2i > n, where n is the number of nodes.
If 2i > n, node i has no left child.

And if you want to know what the child of the node 3 is the left child of the node 3 has the label 6 and in this example the left child over the node 15 is should have been 30, but that label is more than the total number of nodes in the tree, and therefore the left child of node i is 2 i unless 2 i is more than n. In this case, 30 would be more than 15 and therefore, the left child is not well defined. If 2 i is more than n, then i has no left child, as in the example of 15, 14 all the way up to 8.

(Refer Slide Time: 09:17)

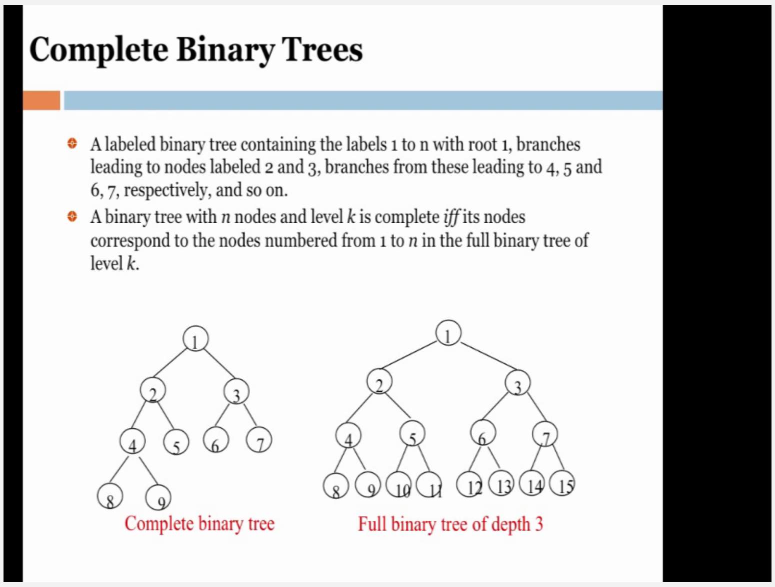


Node Number Properties

Right child of node i is node 2i+1, unless 2i+1 > n, where n is the number of nodes.
If 2i+1 > n, node i has no right child.

Same with the right child, the right child of the node i is 2 i plus 1, unless 2 i plus 1 is more than n. In other words, for any number which is greater than or equal to 8, there is no right child and this is encoded in these labelling schemes. One major application of these node numbering properties is that these give you a convention of storing a binary tree in an array.

(Refer Slide Time: 09:56)



Complete Binary Trees

- A labeled binary tree containing the labels 1 to n with root 1, branches leading to nodes labeled 2 and 3, branches from these leading to 4, 5 and 6, 7, respectively, and so on.
- A binary tree with *n* nodes and level *k* is complete *iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of level *k*.

1
2
3
4
5
6
7
8
9
Complete binary tree
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Full binary tree of depth 3

Especially, if you want to represent a complete binary tree in an array, the root can be stored at the first element, the left child of the root can be stored at the second node in the array and the right child of the root can be stored at the third element of the array, the fourth element of the array contains the left child of the element in the second element of the array, second location on an array and the fifth location in the array contains the right child of the element of the second location of the array.

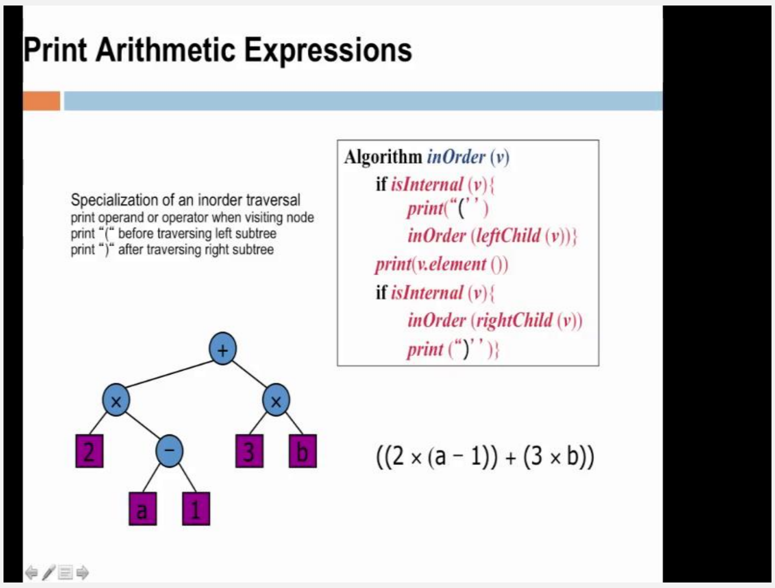
Observe that the most important thing here is that the root must be given the label of 1. This is the very crucial thing. Therefore, observe that this way of labelling which we have studied in the last couple of slides is not a very, not a trivial useless kind of a labelling, it is very important. If you want to implement a binary tree in an array, then you know exactly where in which locations these specific nodes have two set.

And we also have a convention that such a tree is a complete binary tree. That is a complete binary tree is one in which all the nodes at a particular level have the labels which are in an increasing order from the smallest label available for that particular level. That is the binary tree with n nodes and level k is complete, if and only if all the nodes correspond to the numbers from 1 to n in the full binary tree up to level k.

Observe that if you have taken the nodes 8 and 9 and make them children of the node 5, then you would have not had a complete binary tree, this is the most important thing. You take a full binary tree and eliminate some of the nodes which have the largest labels, consecutive ones that is from 12 to 15, you remove all of them, then you get a full binary tree with 11 nodes in it. If you eliminate the all the nodes labelled 8 to 15, then you will get a full binary tree of depth 2 now and so on and so forth.

So, note the difference between a complete binary tree and a full binary tree and specifically note, importantly the definition of a complete binary tree. This way of viewing complete binary tree and full binary tree immediately suggest to us, a very natural implementation of a complete binary tree in an array. The most important thing is that a root element of the complete binary tree is stored at index 1 in the array and the remaining elements are stored at appropriate multiples. For example, the location 2 contains, the index 2 contains the left child of the root element, the location 3 contains the right child of the root element and so on.

(Refer Slide Time: 13:09)



Print Arithmetic Expressions

Specialization of an inorder traversal
print operand or operator when visiting node
print "(" before traversing left subtree
print ")" after traversing right subtree

```
Algorithm inOrder (v)
    if isInternal (v){
        print("(' ')
        inOrder (leftChild (v))}
    print(v.element ())
    if isInternal (v){
        inOrder (rightChild (v))
        print (")' ')}
```

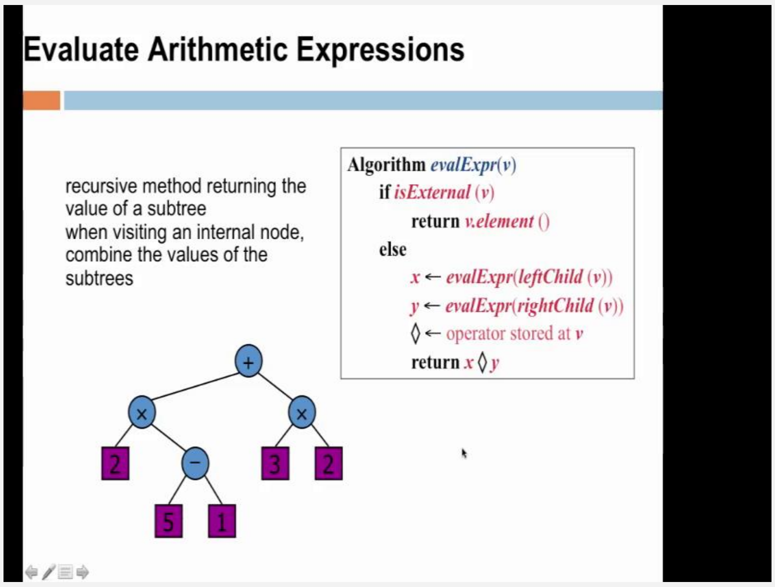
$((2 \times (a - 1)) + (3 \times b))$

Let us look at the exercise of printing arithmetic expressions. So, what we do is that we follow the following simple rule that we place a parenthesis just before traversing the left sub tree, and print the close parenthesis after traversing the right sub tree, and print the elements in order, then we will be able to print the expression exactly in the order in which we would want to implement it in the using the board mass rules.

So, let us try this, so I put an open parenthesis here, I put another open parenthesis here, then I print 2, then I print star that is to print this one, now I put an open parenthesis here print a, print the minus 1, print 1, then close the parenthesis, then close the parenthesis, then I go here print plus, then go to the right child and recurs, then observe that the expression is printed exactly in the order in which I would wanted to be evaluated.

That is the innermost parenthesis gets evaluated first, then among the innermost available innermost parenthesis the one left gets evaluated, the multiplication is done, then this parenthesis gets evaluated, the result is added and that is considered to be the value printed by the tree that is for expression evaluation, but this is for printing the expression.

(Refer Slide Time: 14:55)



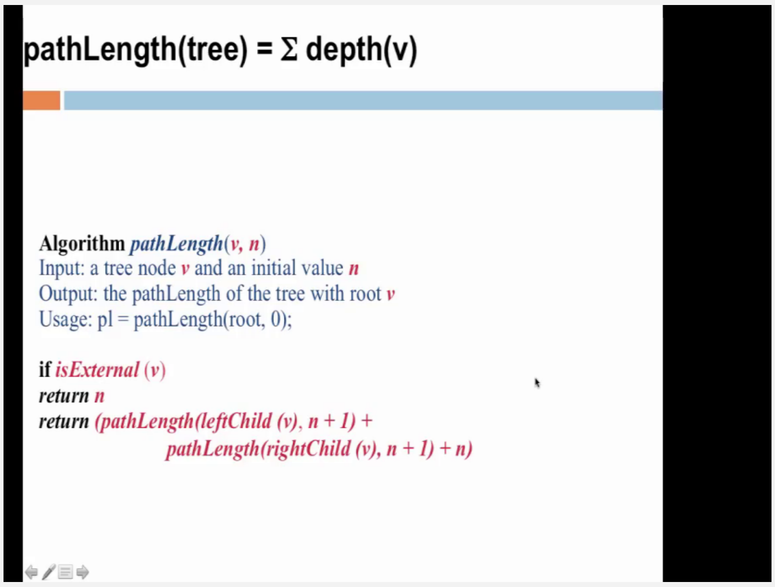
Evaluate Arithmetic Expressions

recursive method returning the value of a subtree
when visiting an internal node, combine the values of the subtrees

```
Algorithm evalExpr(v)
    if isExternal (v)
        return v.element ()
    else
        x ← evalExpr(leftChild (v))
        y ← evalExpr(rightChild (v))
        ◊ ← operator stored at v
        return x ◊ y
```

How does one evaluate the arithmetic expression? Exactly as we did just now, so you evaluate the left child, evaluate the right child, you pick up the operator, return the results of the left child and right child evaluation and print out the value.

(Refer Slide Time: 15:19)



pathLength(tree) = Σ depth(v)

```
Algorithm pathLength(v, n)
Input: a tree node v and an initial value n
Output: the pathLength of the tree with root v
Usage: pl = pathLength(root, 0);

if isExternal (v)
return n
return (pathLength(leftChild (v), n + 1) +
            pathLength(rightChild (v), n + 1) + n)
```

Here is the last application which called the pathLength of the tree. The pathLength of the tree is very important and understanding. Say for example, how much of wire is required to represent this particular tree if you view it as a Boolean circuit? So, the pathLength of the tree is essentially the sum of the different depth, this is the definition. So, you look at all the nodes, look at the depth of the every node and add up all the depths, we define this as the pathLength of the tree.

It is very easily defined as a recursive function let us just take a look at this. If it is an external node and if you are doing a pathLength on a tree with depth n and if the node is at depth n then you return the value of the depth. The usage is that the function gets called with the root node given the value zero. If it is not an external node, we return the value which is the pathLength from the left child at n plus 1 plus pathLength of the right child at n plus 1 plus n and end.

This whole thing is evaluated first and then the value of n is added to it that is n is a value of the parent also. So, it is an important exercise for you to now check that this implementation of pathLength exactly computes a sum of the depths that is the exercise. So, let us see we want to write a recursive function, this function is called pathLength and the result of this pathLength when called at a root with an initial value zero is a request return the sum of all the depths and the rule is very simple.

If it is a leaf node, you return the depth value itself and if it is not a leaf node, you go to the left child, increase the depth value is n plus 1, go to the right child increase the value depth plus 1, compute the pathLengths there, then add the pathLength value n and return the value. With a pen and paper exercise, you can easily check that this indeed returns a sum of the depths of all the nodes in the tree. However, I encourage you to write this as a recursive function in C and check that it works.

So, this brings to an end, this set of three lectures on the tree abstract data type and this is our first explosion on this course to a non linear data structure. Observe that all the concepts that we have studied in the earlier data types are also implicitly used. For example, when we use recursion you are using a stack. When you maintain the left most child, right sibling notation, remember that we are maintaining the doubly linked list.

And when you traverse these nodes when you perform a traversal, let us say a postorder or preorder traversal of a tree, then the children are traverse in a queue like order from the, in the order in which the data is represented in memory. Therefore, the algorithmic methods associated, the methods associated with a tree abstract data type and the binary tree data type implicitly have use all the methods that we have studied with the other data types.

And you can observe that the purposes of the binary tree and a tree in general are many a binary tree for an example is an order tree, you can represent the arithmetic expressions using a binary tree, because of the factor that is an order tree and you can evaluate these expressions in the appropriate way following the board mass rules and get consistent answers. So, the tree abstract data type is an extremely important concept that you have learnt over this week.

In the next week, we will see how to argument additional properties of the data on to the binary tree and implement what are called binary search trees and heaps and using heaps we will implement priority queues. Remember that we studied queues a few lectures back. So, that brings to an end this set of lectures for this week, and when we meet next we will be studying more sophisticated data structures and their implementations.

Thank you. Hope you have enjoyed these lectures.