

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

**Lecture – 14**  
**Tree ADT and Traversals**

Welcome to this week 7 of lectures in this programming and data structures course. So, this week we are going to explore the story of non linear data structures. Recall that over the last 6 weeks we started off with the role of programming, the role of recursion, writing recursive programs and understanding the role of recursion and then we studied linear data structures. We have now written three programs, I mean each of you have submitted three programming exercises and essentially using linear and almost linear looking data structures, and it is time to move on and look at non linear data structures which are very powerful which play a very central role in computer science.

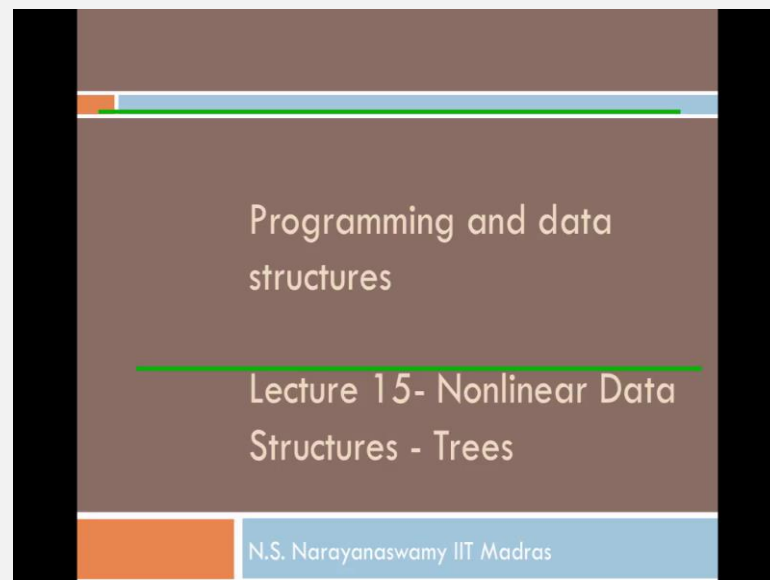
So, in these lectures this week what we are going to do is we are going to look at the tree abstract data type, and once we will look at the tree abstract data type, we will also look at what the primary usage of these data types are and we will look at traversal algorithms, so if you want to access data in a tree, you have to traverse the tree which means you have to go from one node to another node from a fixed starting point which is called the root. And then we will look at special kinds of trees which are very important for designing other data structures that we will study during this course over the next couple of weeks.

And we will also as usual look at some amount of code and you would already seen that over the last couple of weeks, the code is actually fairly simple I had code data inside the program, just to give you a feeling of the compilation and the execution and in any case we look at the implementation of the different methods that we talk about associated with every data type. We will do the same thing with tree data type and the binary tree data type, and we will see what distinguishes that two of them and what are the different concepts that you will need to keep in mind when you think of a binary tree and when you think of a tree.

So, what we do now in this first lecture this week is to look at the tree data type, and then we will look at the implementation of the methods associated with the tree data type and

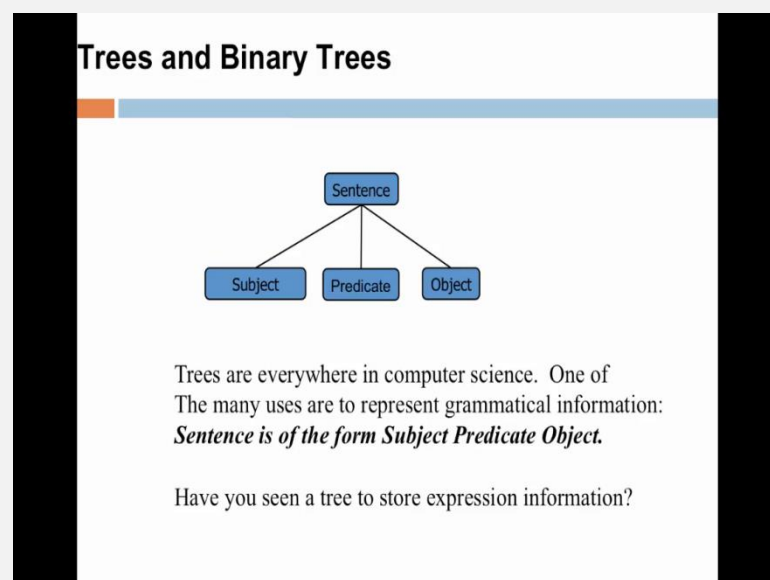
a simple programming exercise. The implementation of the programming exercise we will visit again about a couple of lectures from now. So, let me now go to the presentation.

(Refer Slide Time: 02:48)



So, we are going to talk about non linear data structures, may I go to start off with the simplest of these non linear data structures that are trees.

(Refer Slide Time: 02:59)



We are going to look at trees and binary trees, I would not define a binary tree or a tree now, but I will give you an example. The primary role of this data structure tree is to

typically represent some form of grammatical information. So, when I mean grammatical information, the structure of the tree says something about the values which are associated with the nodes of this tree. So, as you can see, the tree has a structure, the tree has nodes, the nodes have values and the structure, we will see what the structure of the tree is tells us something about the relationship among the values which are in the nodes of the tree.

The best way to do this is a start off with the simple example and in all almost all spoken language is this grammatical structure, the grammatical structure of what constitute it is sentence is very important. For example, in the English grammar a sentence is an object and we know that every sentence has a subject, a predicate and an object and the predicate relates the subject and the object. For example, we can say Rama married Sita, so clearly this is a sentence and we saw that Rama is a subject and the predicate is married and the object is Sita.

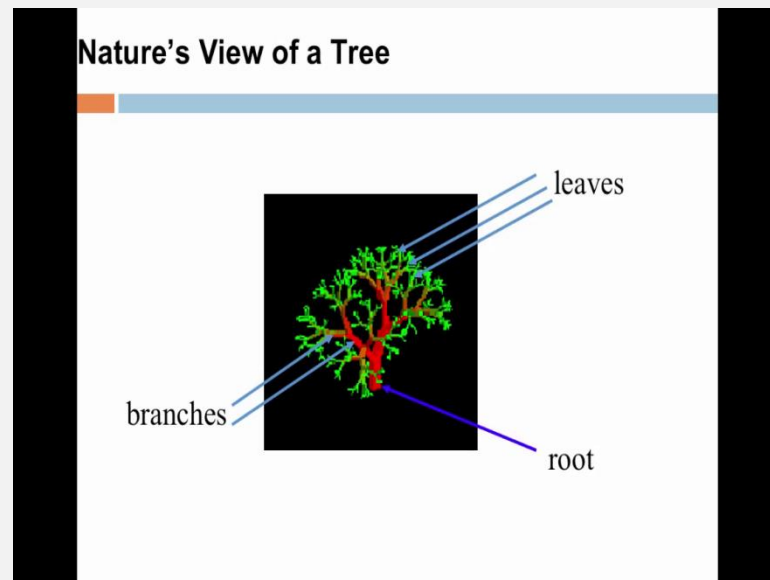
So, this is the very simple example and you can see that this pictorial representation that represents the rules associate with the sentence is indubitably appellat. So, this picture in essence tells you that every sentence has a subject, a predicate and an object. So, therefore one can imagine that this data structure stores the grammatical information associated with the concept called sentence and the grammatical information regarding the concept sentence involves the subject concept, the predicate concept, and the object concept and we refer to this pictorial object as a tree.

Over the coming set of lectures, we will formalize the definition of a tree, we will also visualize the tree in a certain manner and we will also see how to write programs which can create trees as in memory data objects that is data objects which just sit in the memory during execution and how we will manipulate the trees, how we will actually visit different nodes of the trees and so on and so forth, so that is a focus.

So, what are trees? Trees are abstract objects, they in some sense look like this. They have nodes, each of these is called a node and there is a values sitting inside a node and there is a relationship between the nodes and this relationship also imports a relationship between the values which are sitting at the nodes, the values that have been placed at the nodes. We will get more precise as we go through and we will also see how to define these trees in a programming language and while execution, how to visualize the tree as

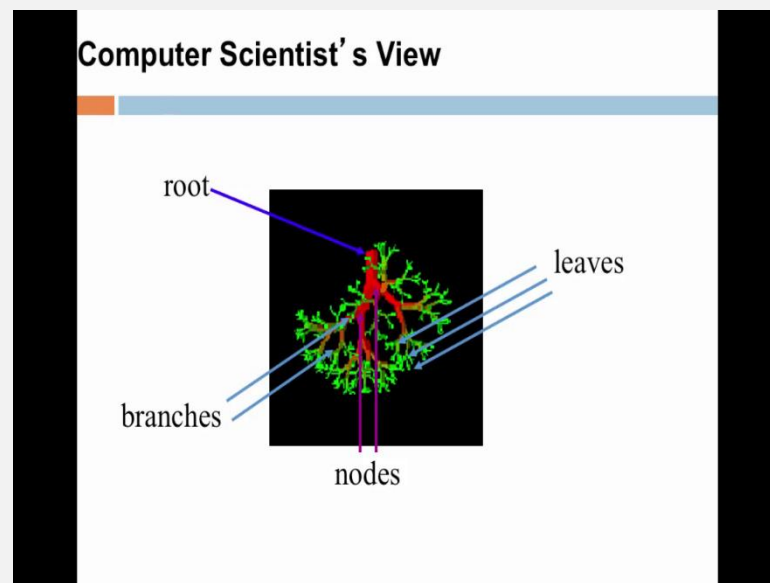
a run time object manipulated by the program. These are the things that we are going to study when we talk about trees.

(Refer Slide Time: 06:38)



Of course, the word tree is not has not originated from Computer Science, Computer Science is much younger than the concept of trees and if you look at, if you ask a non computer scientist what a tree is, he or she will immediately agree that this is a tree, it has a single root, it has multiple branches. In this case, I have drawn like two arrows to indicate that more branches than the number of roots typically and three arrows to indicate that there are many more leaves than the number of branches and definitely more number of leaves than the total number of roots.

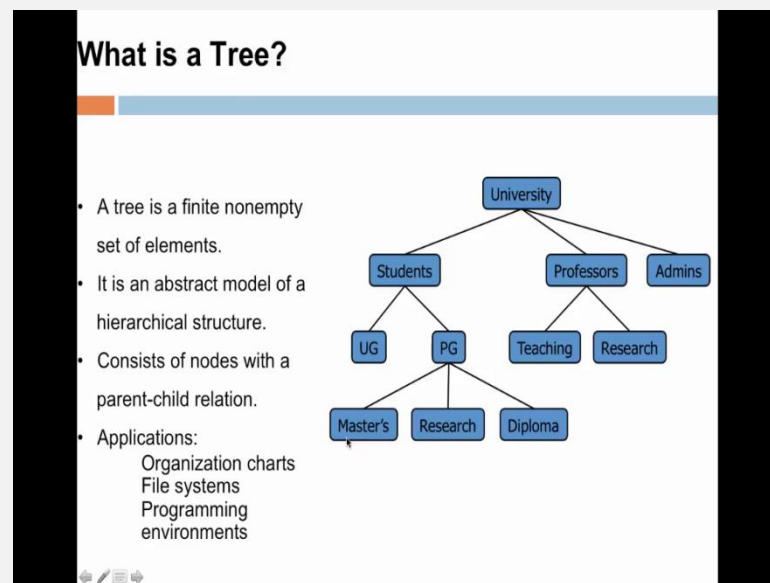
(Refer Slide Time: 07:28)



So, this is the nature's view of a tree and this is a computer scientist view of a tree. It is upside down, there is a single root and there are multiple branches, but we also introduce the concept of nodes, nodes are the places where the branches branch off. As you can see, this pink lines or violet lines actually go here and this is a node and to the left is a branch, to the right is a branch into the left is a branch and the furthest entities from the roots are really called leaves.

That is places from where there are no further branches or a special kinds of nodes and they are called leaves. Now, what can you do with this computer scientist view, one can actually keep track of the root and keeps some values at the nodes, keeps some values at every node, keeps some values at the trees, keeps some values at the leaves and then manipulate this object. For example, we can eliminate some leaves then we can add a few new nodes, we can add a few new leaves and so on and so forth. This is a computer scientist motivation for viewing this data object. Most importantly the concept that you should keep in mind with respect to the computer scientist view is that the trees seems to keep some hierarchical information that is the most important thing.

(Refer Slide Time: 09:03)



So, what is a tree? As far as computer scientists are concerned, it is a finite nonempty set of elements. First of all it is finite, there are some number of elements definitely at least one element. It is a hierarchical object, so it is a model of hierarchical object and the elements are called nodes. That is the elements in a tree, if somebody asks you what are the elements in a tree, the answer is that these are nodes, the nodes have a position and any pair of nodes have, so for every node there is a well defined parent or a child relationship with another node.

So, what are the typical applications of trees? These are ways of representing organization charts and some systemic information, the directories structure on your computer is also a classic example of a tree and so on and so forth. So, let us look at this picture, what is this picture saying? This picture imports some information about a concept called university, so university is a value which is sitting inside this node and this node we are going to refer to as a root node.

A university has students, a university has professors, a university has administrators, students are of two kinds under graduate students and post graduate students and post graduate students are of three kinds. Students, who are studied from post graduate diploma, students who are engaged in post graduate research and students who are obtaining a masters degree from a post graduate study.

Professors are also further classified into two kinds of people using the concepts of

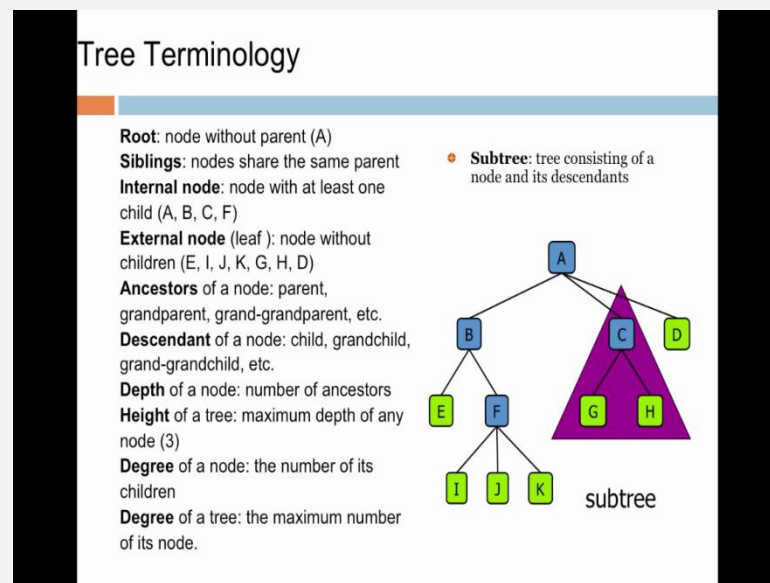
teaching professors and research professors and these are the concepts that we have focused on, but observe that apart from the concept, there is the root node, there are three children of this root node and this node has two children and you can think of these two children as a grand children of the root node, and similarly these two children as a grand children of the root node and children of this particular node.

Observe I am very carefully avoiding saying that UG is a descendent of university. That is a technically incorrect statement to make, the descendant and ancestor relationship that you might have studied are the one that I am going to tell you in the next couple of slides is a relationship between the nodes and not the values. Of course, it is possible to visualize and it is indeed correct to say that the value at a descendant is also a descendent of the value at the particular ancestral node.

In other word, it is indeed not completely incorrect to say that UG is a descendant of university, but if you try to understand that sentence in English, it does not make sense. So, therefore we will avoid coming up with ancestor descendant relationship among the values, but we will always use ancestor descendant relationship as structural relationships, so this is what a tree is. So, let me summarize again, a tree has at least one element and the elements are nodes, the nodes contain values, the nodes contain data and every node has a parent or a child relationship with some other node.

Of course, some nodes have both parent and child relationship, for example this node has a parent and this node even has a child. Of course, this node does not have children, but this node has a parent. The root node does not have a parent, but has children. Now, observe that I have try to communicate to you an abstraction of this object called a tree, how to use this, it is left your imagination. And we will definitely look at a few examples and we will use them in our programming exercises. As you can see, this is one example which uses the abstraction of a tree.

(Refer Slide Time: 13:32)



Let us look at the terminology that I spoke about in the previous slide, so a root node is one that does not have a parent. Sibling nodes are those that do not have... Sibling nodes are those nodes which share the same parent, our internal node is one that has at least a child and an external node otherwise called a leaf node is one that does not have children, the ancestors of a node which are the parents, the grandparents or the great grandparents and so on and so forth are examples of ancestors of a node.

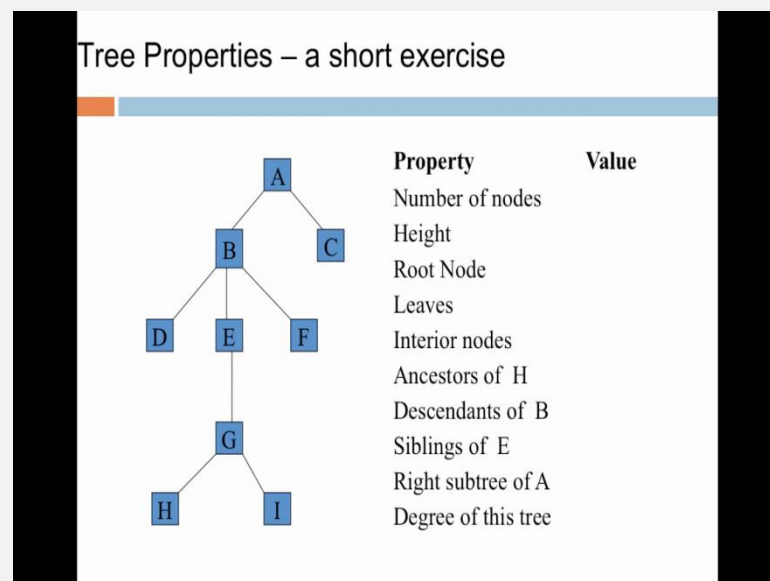
The descendant again is a child, the grandchild, the great grandchild are all examples of descendants of a node, the depth of a node this is a numeric parameter associated with every node. It is a number of ancestors that the node has, the height of the whole tree is the maximum depth of any node. In this example, we will observe that again it is a numeric parameter, the height of the tree is 3, the root is at level 0, this node is at level 1, this node is at level 2 and this node is at level 3.

And therefore the height of, the depth of a node is depth of this particular node, the node that contains the value k is actually 3. The degree of the node of a node is a number of it is children in this case and the degree of the tree is the maximum degree of it is nodes. So, I have written something that is incorrect here, so not the maximum number, the degree of a tree is the maximum degree of it is nodes. Practices sub tree of this tree, the region in violet is a sub tree of this tree, it is a tree which consists of a node and it is descendants.

So, let us do some small exercises here. You can see that E, I, J, K, G, H and D are all external nodes, otherwise called leaves, because they do not have any children. A, C, B and F are internal nodes, they all have children. A is a special node it is a root, it does not have parent. C and D are siblings, G and H are also siblings, actually B, C and D are siblings, K is a descendant of B, K is a descendant of F, K is also a descendant of A, K is not a descendant of C.

And of course, one can say that K is a descendant of sibling of C which is correct, but K is not the descendant of every sibling of C, that is K is not the descendant of D, but it is a descendant of B and so on and so forth. So, as you can see in this slide, the most important concept is not the values in the node. It is about the structural relationship that is among the elements of a tree, the elements of the tree are the nodes. The nodes store values that we will come to short form.

(Refer Slide Time: 16:59)



So, let us look at the short exercise about the tree properties, let us ask how many nodes are there in this tree. As you can see, there are 6 plus 3, 9 nodes. What is a height of the tree? The height of the tree is that value which is the depth of the deepest node. So, let us look at the depth of the deepest node, A is at depth 0, B is at depth 1, E this node is at depth 0, this node is at depth 1, 2, 3 and 4. Therefore, the height of the tree is 4. The root node is this node that is a node that contains a value A.

The leaf nodes as you can see the nodes that contain D, the nodes that contain H, the

nodes that contain I, node that contains F and the node that contains C. The interior nodes as you can see are those which are not the leaves. Now, similarly you can write down the ancestor of the node that contains H. So, the ancestor of the node that contains H are all these nodes, right from the nodes that contain A, the node that contains B, E and G and so on and so forth. So, it is a simple exercise and since we would have access to the slides and we will also put up a small exercise with this main function.

(Refer Slide Time: 18:16)

## Tree ADT

We use positions to abstract nodes

Generic methods:

- integer **size()**
- boolean **isEmpty()**
- List **elements()**
- List of nodes **positions()**

Accessor methods:

- position **root()**
- position **parent(p)**
- positionIterator **children(p)**

- Query methods:
  - boolean **isInternal(p)**
  - boolean **isExternal(p)**
  - boolean **isRoot(p)**
- Update methods:
  - swapElements(p, q)**
  - object **replaceElement(p, o)**
- Additional update methods may be defined by data structures implementing the Tree ADT
- Delete(p) – maintain the tree
- Insert(p,q) – insert p as child of q and maintain the tree.

Now, we are ready to talk about the specific data items that are associated with the tree and the methods associated with the tree. So, let us look at that methods, the methods are the size of the tree, a Boolean query is to whether this is the tree is empty, we now make a very precise distinction between the elements on the positions. So, when I say elements, the method returns the values which are at the nodes and the method position returns a pointer to the list of all nodes.

The accessor methods are it returns a position that is when you ask for the root of a particular tree, you get a pointer to the root of the tree. The parent method takes as an argument a pointer and it returns a position which would be the parent of p. Of course, it should return an invalid query kind of a message, if p is given to be the root, because the parent of p is undefined. We will come up with some conventions here.

The children of p would return a pointer to the children of p, that is this would be a, the return value of this method is a pointer to a list of nodes that is what I written as a

position iterator. The query methods are queries like is a node, an internal node or is it an external node or is it a root. These are very easy to check if the parent is not defined for a point for a node, then it is a root. If a node does not have a descendant, it is an external node or leaf otherwise, it is an internal node. The root is also an internal.

Here are some update methods, we swap elements p and q. Now, remember that the elements of the values which are in the nodes therefore, when we swap elements p and q the values in the nodes p and q are exchanged. The replace element at a particular node by an object o returns a value whether, depending on whether the object was successfully replaced or not. There are no limitations to the kind of methods that we include with any abstract data type and this is more true more, so this is very true with the tree abstract data type, because multiple users in the tree abstract data type come up with multiple backups.

The delete method deletes a node from a tree, but it must also maintain a tree. Similarly, insert a node into the tree as a descendant or inserted as a child of q. That is insert the node p as a child of q and maintain the tree structure, these are all natural methods which update the structure of the tree. So, as you can see these are generate methods which ask for the values, ask for the positions, ask for data associated with tree.

These are accessory methods which tell you information about whether a node is a root of the tree or whether and what it is parent and children are. These are query methods which extract properties of different nodes in the tree and here are the update methods which update the values which are stored in the tree. So, therefore over the last few slides, over the last 15 minutes we have looked at pretty much everything that we need to know about the tree abstract data type to be able to use it.

(Refer Slide Time: 22:34)

The slide is titled "Intuitive Representation of Tree Node". It contains a section "List Representation" with a bullet point showing a nested list:  $(A(B(E(K,L),F),C(G),D(H(M),I,J)))$ . Below this, another bullet point states: "The root comes first, followed by a list of links to sub-trees". A diagram below the text shows a horizontal bar divided into segments: "Data", "Link 1", "Link 2", an ellipsis "...", and "Link n". Above the "Link 2" segment, a blue thought bubble contains the text: "How many link fields are needed in such a representation?".

Let us look at an intuitive representation of a tree. So, one of the most popular ways to visualize a tree is the list representation. So, let us look at this particular list, as you can see the list contains further list inside it, so this whole list has multiple list inside it. So, let us look at the multiple list, for example K comma L is a list, just the element M is a list. Now, you can see also this list whose elements are H, the list containing M, then the element I and the element j.

That is the element in this particular list or the element H, the list containing the element M, the element I and the element J. So, now I am going to specify a tree using the recursive specification and it is specified as follows. In a list, the root is the first element of list, so every list now we are going to visualize, now if a list is properly parenthesis, then the first element of the list is the root of the corresponding tree.

The remaining elements of the list are the children and the sub sequent descendents of the tree, of the root node, let me say this again. Whenever you see a list which is properly parenthesis, the first element is the root of the tree that is represented by the list. The remaining elements in that list are the children of the root node again specified recursively, so this rule is applied recursively. So, let us look at this example, A is a root node of the tree and now let us move towards the children.

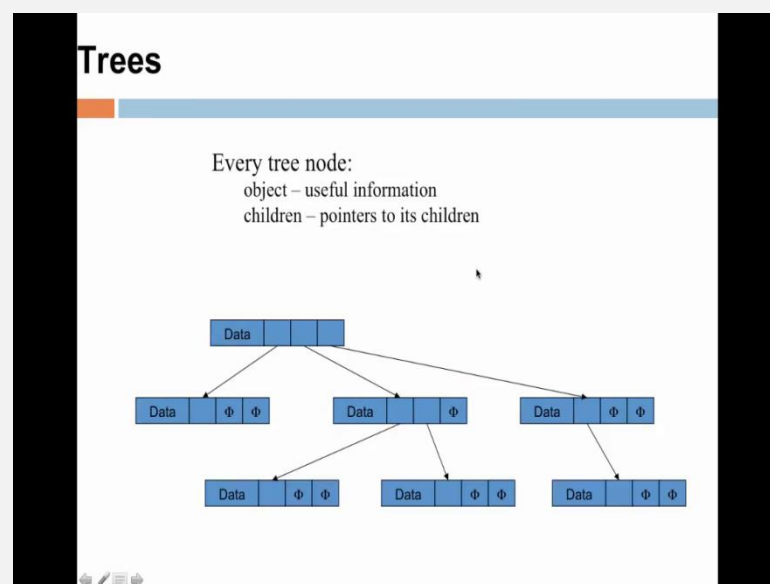
So, this parenthesis starts here, and let us see where does it end. This is matched by this, and this matched by this, this is matched by this, this is matched by this, then this is

matched by this, now you can see that this parenthesis matches this parenthesis. Therefore, A is a root without single child and that is single child is a node B and everything else is at descended of B. It is either a child of B or a descended of B.

A is a ancestor of the node B or a parent of the node B and A has exactly one child in this example and it will always turn out that every node has a unique parent or zero parents. As you can see, every properly parenthesis list, I should not say properly parenthesis, but I am saying this for redundancy. Every list is indeed representative of a tree and how does one view a list as a tree. One view as a list a tree by thinking that the first element of the list is the root of the tree, and the descendents of the root are given by the remaining elements in the list.

So, now we have looked at an intuitive way of thinking of a tree. Of course, this can immediately be converted also, not just is this representation an intuitive representation, it is also a representation that can be converted into an implementation. So, using the linked list concept you can define a tree very precisely by recursively defining a list of lists.

(Refer Slide Time: 26:43)

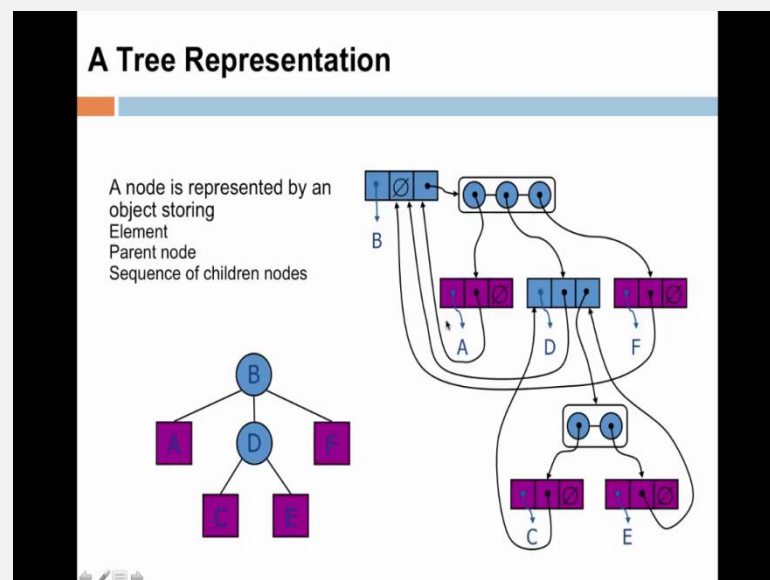


Now, let us just look at what one can do with the tree in an abstract setting. So, the tree consists of nodes, otherwise called positions. Every node contains an object which is the value or the useful information that is the data item in this picture and then it contains pointers to its children. So, this is essentially the abstract view of a tree, when you think

of it, it does not in memory object. The earlier pictures of trees ((Refer Time: 27:17)) were our picture that are imagined by people across multiple domains.

So, for example a person who is maintaining information about the university structure may not be a computer programmer, but can still think about the information being organized in a tree. Similarly, a computational link list or not necessarily a computational link list, the persons studying any language can think of a tree to be this particular object. A link list is in memory representation and this is the programmer's imagination of a tree. So, this is a node, this is a record or this is a structure, there is one field to store a data and in these examples there are three fields to store pointers. The pointers point to different children of this particular node.

(Refer Slide Time: 28:18)



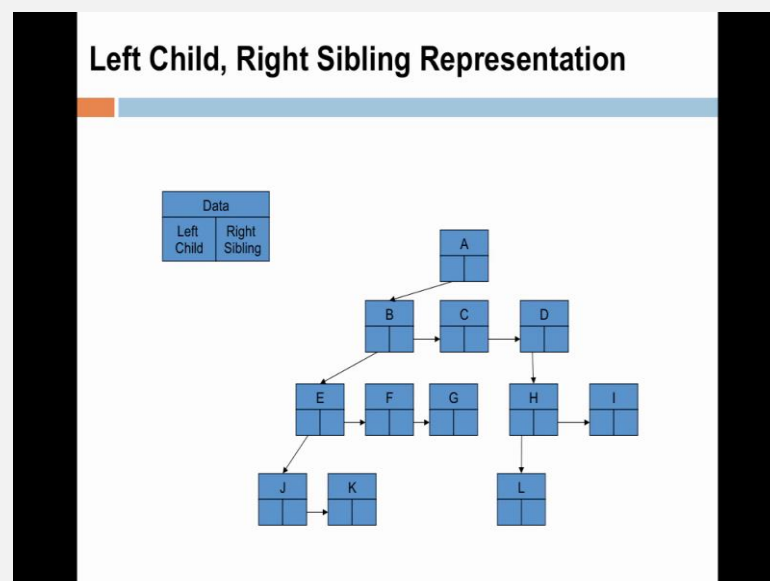
Here is a tree like, a tree representation which is ((Refer Time: 28:26)) just slightly more an elaborate way of representing the picture in this particular slide. So, every node in the trees represented by an object, the objects stores an element, the element is a value. In this case, B is a value, A is a value, D is a value, F is a value, E is a value and so on, then the node has a pointer to it is unique parent. So, in this case this pointer goes to the parent, in other words this node which is violet in color containing the value A as a unique parent which is this node which is completely color blue and contains the value B.

Observe that the parent field of this particular node has this empty set symbol. This

basically says that this node is a root. Now, this node as you can see has some information about the descendants of D, in other words the children of B are represented by this particular data object which has pointers to this node, this has a pointer to this node, this has a pointer to this node and so on. So, in particular you can now see that the node that contains D has two children and these two children are visualized here and there are pointers to the corresponding nodes.

Therefore, these white boxes are pointers to nodes and the color boxes are nodes themselves. You can think of this as the in memory data structure, you can visualize these pointers in this passion. This arrow mark likes, you imagine the value which is here, the other arrow marks kind of, our pointers to other nodes, these are addresses and s on and so forth. So, the abstract picture of this tree which is the in memory data structure is this particular object. So, there are as you can see six nodes and the values inside the nodes, the value stored by the nodes are the values which are written inside the nodes in this abstract object.

(Refer Slide Time: 30:55)



The previous slide actually was a realization of what is called the left child right sibling representation of a tree. So, observe that A is a parent node, there is the pointer to the left child which is another node which contains a value B and B has a pointer to it is child, but B itself has a pointer to it is right sibling. Sibling is remember, B and C are siblings, if they share a common parent. So, B keeps a pointer to it is right sibling, C keeps a pointer to it is right sibling.

Therefore, it is unambiguous that A is the root two which has three children which are B, C and D, A remembers B and B remembers C, the right sibling and C remembers D, it is right sibling. Now, D has a child which is called H, H has a child which is called the L, L does not have any right sibling, therefore H has only one child. D has H as the child and H has one right sibling which is called I, therefore B and I are the two children of... H and I are the two children of D and so on and so forth.

So, this is called the left child right sibling representation of a tree. Therefore, if you done the program this, we will definitely end about a program the left child right sibling representation and when you write a programs and if you are maintaining a tree, the left child right sibling representation is a good way to represent a tree. Actually in our implementations we will also, though we have not mention this, there will be an additional fields here for storing the parent pointer which we have not listed here, but when you see the implementation, we have done that also.

Actually we here also maintain a doubly linked list that is not only does, we know that C is it is right sibling, C also knows that B is it is left sibling. This is to ensure that we use all the power of the linear data structures that we have learnt in the previous lectures.

(Refer Slide Time: 33:17)

**Tree Traversal**

Two main methods:  
**Preorder**  
**Postorder**

Recursive definition

**Preorder:**  
visit the root  
traverse in preorder the children (subtrees)

**Postorder**  
traverse in postorder the children (subtrees)  
visit the root

So, what have we done so far, so we have really answered questions about what is a tree. Tree has structural relationship among nodes. What are the nodes there for? The nodes are there to store values. How does we actually visualize a tree? You can visualize a tree

as an abstract object or you can visualize a tree as an in memory object. When you think of the tree as an in memory object, you think of the values which are sitting at different fields in the structure associated with our node and some of these values would be addresses, in other words pointers and nodes are classified into different types.

There is one unique node which is called a root node, there are multiple trees, there are many multiple leaves and multiple internal nodes. Now, one can actually ask the question what can one do with a tree. Of course, just in a jokuller manner of speaking, what can you do with a biological tree, when you climb it? You can extract fruits from it, you can populate the tree with more fruits by giving it more nutrients, by giving it the better environment flower and fruit.

What do you do with a tree as a computer scientist, you can store values in it, you can retry values from it, you can add nodes to the tree, you can delete nodes from the tree, you can answer questions about what values are there in the tree, you can answer questions about how many nodes are there in the tree, you can answer questions about who is a precider, who is an ansistor, who is descendant of a particular node in a tree and so on and so forth.

For all these operations of a tree, you should be able to climb the tree or climb down from the tree, if it was a biological tree, if we have to traverse a tree you should be, if you want to able to manipulate the tree data structure we have to be able to traverse a tree. Now, you can see that traversal is a common theme associated with data structures. Whenever you set up a data structure you should be able to traverse the data structure, you should be able to move to a particular node and access the data item there.

You should be able to modify the data item, you should be able to check if a certain data item occurs at some node in a tree therefore, or there in the data structure. Therefore, central to data structures is the issue of traversing the data structure, traversing meaning moving from one pointer, one node to another or moving to a specify node and so on and so forth. Therefore, for tree traversal there are two special methods two main methods called the preorder traversal and the postorder traversal.

These are two methods which are different from the other methods that we have seen in the tree ADT, of course these are also part of the tree abstract data type. These are called the preorder traversal method and the postorder traversal method. These methods are designed recursively and because these are traversal methods, the primary activity of a

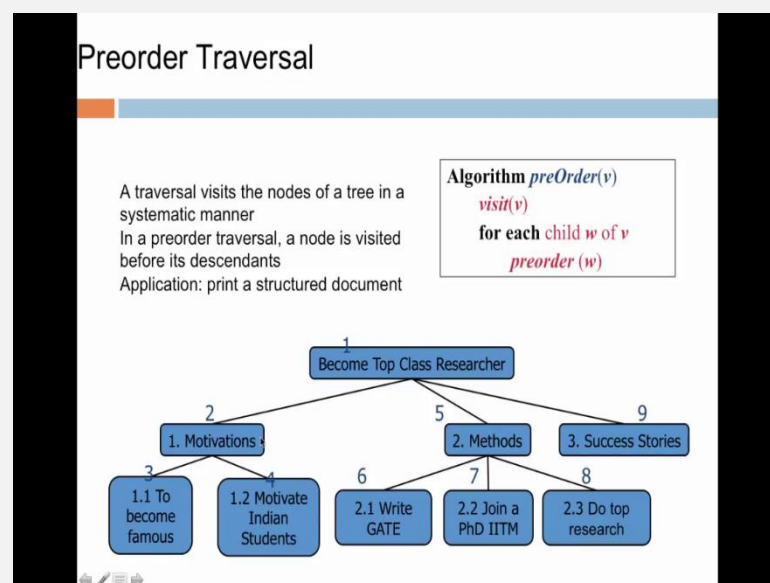
traversal method is to visit a node. The preorder and postorder traversals tell you in what order to visit different nodes in a tree and these are specified recursively.

So, let us see what the preorder traversal says. The preorder traversal of a tree which is given access to the root pointer, so let me say this again. The preorder traversal rule takes an argument a pointer to the root node in the tree, it visits the root node then it traverses in preorder the children that is, it does a preorder traversal of the children sub trees of the root tree, the root pointer.

And this recursion as you can imagine will go and tell the traversal encounters a leaf, when it encounters a leaf it returns from that recursive call and continues recursively traversing the tree in the preorder fashion, first root then the children. The postorder traversal is interesting you first traverse the children and then mark the root as visited. So, standard confusion of many students how can you in the tree visit the children without going through the root.

Therefore, let me make the technically correct statement, a postorder traversal is at a particular node which is the root is recursively defined as follows. First you traverse the children and then mark the root as visited and then mark the root as visited. So, I hope this is not confusing, so on the other hand in the preorder traversal, when you visit the root node you mark it as visited first, then you go ahead and mark the children as visited in a preorder fashion or mark the dissidence as visited in the preorder fashion. We will see the distinction when we run an example.

(Refer Slide Time: 38:33)

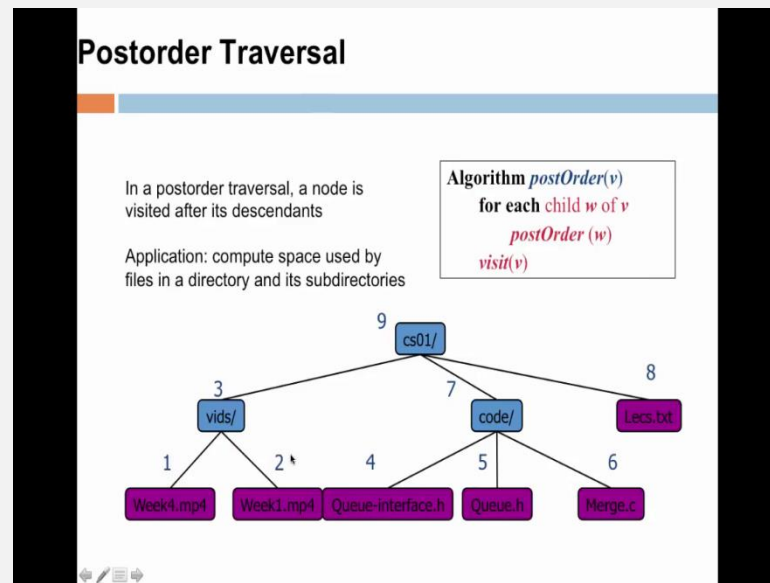


So, let us do this, let us look at the pseudo code and we will also see the order in which the nodes are marked as visited in the preorder traversal. So, the preorder traversal  $v$  is to visit  $v$  and then for every child  $w$  of  $v$ , you do a preorder traversal of  $w$ . So, there is a small typographical error here, there are multiple purposes of preorder traversal right, so what you can do is to print the whole directory structure on your computer using the preorder traversal of the whole directory structure.

So, let us look at this tree, so this tree is probably interesting for any of you who aspire to be a top class researcher. So, if you want to become a top class researcher, your motivation has to be to become famous or to motivate Indian students or may be to motivate and to motivate into your students, you want not only become famous, but you also want to motivate into your students. To become a top class researcher what are the methods you probably write GATE and then you join a PhD at IIT Madras and then you do a top research and then you will talk about your success story.

So, these are all what are done by a top class researcher, so I hope you get the message that I hope some of you become top class researchers. Let us look at the order in which a preorder traversal visits the nodes in the tree, it marks this node first as visited, then two as visited, then three as visited, then four as visited, now it has finished visiting the whole sub tree rooted at this particular node, then it considers this sub tree of the root, marks this as visited then explores this sub tree, then explore this sub tree, then explore this sub tree, then this whole sub trees is visited, then it comes here visits this sub tree, the whole tree is visited and the function exists. So, the numbers associated with every node are the order in which they are marked as visited.

(Refer Slide Time: 40:55)



Let us look at the postorder traversal, the numbering tells you the order in which the nodes are marked as visited. So, this is the directory structure on my computer for this particular course, there is a directory of videos, there is a directory of code, there is a single file called lectures dot text which says what are the lectures that we want to give and then this is the root directory which is the directory associated with the course.

Here are the leaf nodes which are files, which are the videos, these are the different dot h files and this is the dot c file which you have already seen. Let us look at your order in which your postorder traversal visits. Start at the root, you will mark this root as visited only when all the children are visited, only after all the children are visited. Therefore, the first node that gets marked as visited is that node that is visited first and does not have any further children or dissidence that would be this file. It is marked as the first node as marked as visited. Then, this is the second node that is marked as visited.

After if this whole sub trees now visited, so this is the third node that is visited. Now, you observe that this is not still visited, because these three children have not yet been visited, so these leaves get 4, 5 and 6, this get 7 then this gets visited as the 8th node in the tree, then the root node gets visited. So, this is the postorder traversal and really if you visualize the picture it is very hard to forget it. Now, not only can you access the data items in this particular order, you will definitely have this question why should I have these two ways to access the different data items in this node.

(Refer Slide Time: 43:27)

(Refer Slide Time: 43:36)

```

/* This file contains the structures for node and tree */

typedef struct node{
    char data;
    struct Node *parent;           // parent and leftSibling fields are there to make delete easier
    struct node *leftChild;
    struct node *rightSibling;
    struct node *leftSibling;
}Node;

typedef struct tree{
    Node *root;
    int size;                      // size is to maintain the size of the tree
}Tree;

"tree.h" 14L, 379C

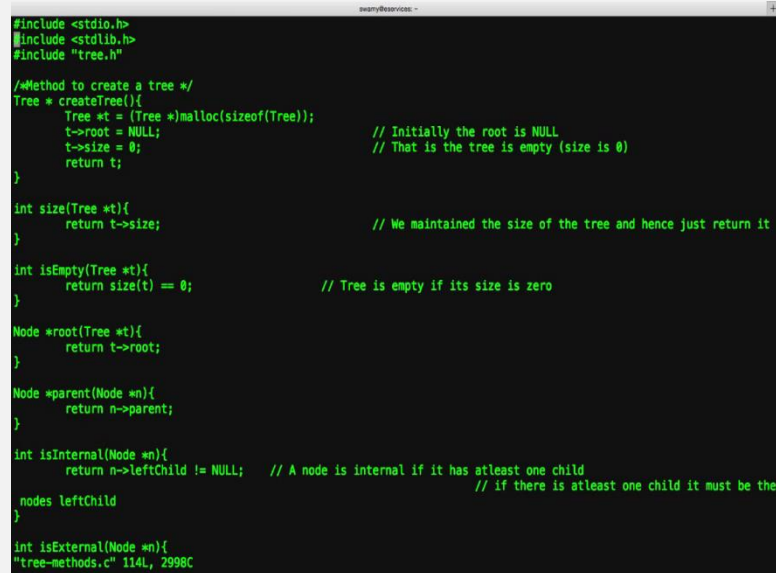
```



will see the code for it, deletes a particular node from the tree and the insert node takes a pointer to a tree, takes a pointer to a node that wants to be inserted and a pointer to the parent. The end result of this is that the node n is inserted as a descendent, the first descendent of the parent in the tree t.

Then, we have traversal methods which are preorder and postorder which starts at the root and traverses the tree in either the preorder or the postorder function. These are the methods associated with the tree. Let us quickly go through the tree interface, implementation of the tree interface methods.

(Refer Slide Time: 46:49)



```
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"

/*Method to create a tree */
Tree * createTree(){
    Tree *t = (Tree *)malloc(sizeof(Tree));
    t->root = NULL;           // Initially the root is NULL
    t->size = 0;              // That is the tree is empty (size is 0)
    return t;
}

int size(Tree *t){
    return t->size;          // We maintained the size of the tree and hence just return it
}

int isEmpty(Tree *t){
    return size(t) == 0;     // Tree is empty if its size is zero
}

Node *root(Tree *t){
    return t->root;
}

Node *parent(Node *n){
    return n->parent;
}

int isInternal(Node *n){
    return n->leftChild != NULL; // A node is internal if it has atleast one child
                                // if there is atleast one child it must be the
                                // nodes leftChild
}

int isExternal(Node *n){
    "tree-methods.c" 114L, 2998C
}
```

So, let us just see what we do in the C programming language, we created a tree it returns a pointer to tree, the way we do it is that we allocate a pointer to the size of the tree. Therefore, a node which is of type tree is created, the root node points to null the size is kept as zero, that is there are no data items there and then you return a pointer to t.

The query to size returns t pointer dot size, this one tells you if the tree is empty or not by evaluating this comparison with the value zero, then this one returns a pointer to the root node of the tree which is always well defined that is why we defined it. Then, we return a node to the parent node, a pointer to the parent node and we check if a node is an internal node, remember that if a node is an internal node then it has no children, in particular because the first of the children is always represented as left child of the node, we just have to check if left child is null. If left child is not null, then you can say that n is an

internal node, if left child is null then you can say that n is not an internal node.

(Refer Slide Time: 48:19)

```
t->root = NULL; // Initially the root is NULL
t->size = 0; // That is the tree is empty (size is 0)
return t;
}

int size(Tree *t){
    return t->size; // We maintained the size of the tree and hence just return it
}

int isEmpty(Tree *t){
    return size(t) == 0; // Tree is empty if its size is zero
}

Node *root(Tree *t){
    return t->root;
}

Node *parent(Node *n){
    return n->parent;
}

int isInternal(Node *n){
    return n->leftChild != NULL; // A node is internal if it has atleast one child
    // if there is atleast one child it must be the
    nodes leftChild
}

int isExternal(Node *n){
    return !isInternal(n); // A node is external if it is not internal
}

int isRoot(Node *n){
    return parent(n) == NULL; // A node is root if its parent is NULL
}
```

Similarly, the external a node which is not, external it is an internal node. Similarly, a root node is one whose parent is null.

(Refer Slide Time: 48:39)

```
}

int isExternal(Node *n){
    return !isInternal(n); // A node is external if it is not internal
}

int isRoot(Node *n){
    return parent(n) == NULL; // A node is root if its parent is NULL
}

/* t is the tree in which this insertion should be done, 'node' should be inserted as child to 'parent' */
void insert(Tree *t, Node *node, Node *parent){ // The tree argument is needed to maintain the size of the tree
    // every time as insertion is done, 'node' is made as the leftmost child of 'parent'
    node->leftSibling = NULL;
    node->parent = parent;
    node->rightSibling = parent->leftChild;
    parent->leftChild = node;
    node->leftChild = NULL;
    // if the parent had a leftChild before this insertion, then its leftSibling is to be updated as the current node
    if(node->rightSibling != NULL)
        node->rightSibling->leftSibling = node;
    t->size = t->size++; // The size of the tree is incremented by 1
}

// void delete(Tree *t, Node *n){
//     if(n->parent == NULL) // if n is the root of the tree
//     {
//         t->root = n->leftChild;
//         if(t->root != NULL)
//         {
//             // 
//         }
//     }
//     else if(n->leftSibling == NULL) // n is the leftChild if its parent
```

So, here we have the implementation for the insert method. Since, you are sharing the code with the, so you can take look at the insert methods. So, I am not really going to look at it, so basically the way to do it, let me tell you logically is that you take the pointer tree, you take the value that you want to insert, you take a pointer to the node, then you add the node as a parent, as a left most child of the parent node and then you

connect the right sibling to the left child of the parent.

So, what you do? You insert the node, how do you insert the node? You basically make it the left child of the parent node, after making it the left child to the parent node ensure that the original left child is now the right sibling of the new left child. So, once you do this, you have null.

(Refer Slide Time: 49:47)

```
//          insert(t, n->leftChild, n->parent);
//      }
//  }
//  else
//  {
//      n->leftSibling->rightSibling = n->rightSibling;
//      n->rightSibling->leftSibling = n->leftSibling;
//      insert(t, n->leftChild, n->parent);
//  }
// }

void preOrder(Node *root)
{
    if(root)
    {
        printf("%c ", root->data);           // first print the root's data
        Node *temp = root->leftChild;
        while(temp)                          // traverse each of the root's
        children
        {
            preOrder(temp);
            temp = temp->rightSibling;        // following the right siblings
        }
    }
}

void postOrder(Node *root)
{
    if(root)
    {
        Node *temp = root->leftChild;        // First the children are traversed
        while(temp)
        {
            preOrder(temp);
            temp = temp->rightSibling;        // following the right siblings
        }
        printf("%c ", root->data);
    }
}
```

So, this is the delete implementation you are welcome to take a look at it, we will talk about that later.

(Refer Slide Time: 49:48)

```
//  {
//      n->leftSibling->rightSibling = n->rightSibling;
//      n->rightSibling->leftSibling = n->leftSibling;
//      insert(t, n->leftChild, n->parent);
//  }
// }

void preOrder(Node *root)
{
    if(root)
    {
        printf("%c ", root->data);           // first print the root's data
        Node *temp = root->leftChild;
        while(temp)                          // traverse each of the root's
        children
        {
            preOrder(temp);
            temp = temp->rightSibling;        // following the right siblings
        }
    }
}

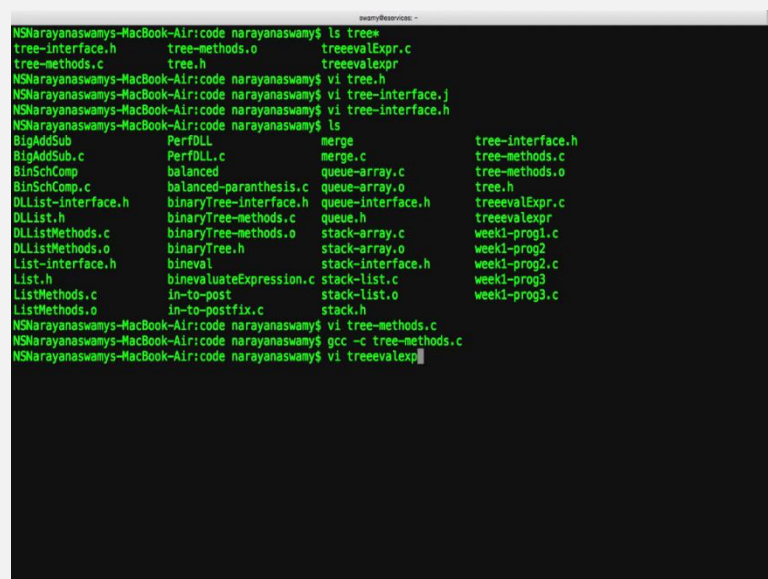
void postOrder(Node *root)
{
    if(root)
    {
        Node *temp = root->leftChild;        // First the children are traversed
        while(temp)
        {
            preOrder(temp);
            temp = temp->rightSibling;        // following the right siblings
        }
        printf("%c ", root->data);           // root's data is printed last
    }
}
```

So, preorder we really have implemented exactly the order in which we just implemented almost exactly what we saw on the slides, it is just converted from pseudo code on the slide to C code in this file. So, if it is the root, you first print the data item there, then what you do is you go to the descendent and do a preorder traversal and you do this preorder traversal by visiting every right sibling starting from the left most node.

So, therefore you look at the left child that is temp and while that is well defined, you do a preorder traversal starting make a recursive call to preorder traversal with temp. Once this preorder traversal is completed then you move to the right sibling of the node and start up your traversal from there. Do this still it becomes null and you recurse, this is the beauty of the recursive specification, what looks like a complex task is represented just in 7 to 8 lines of code. The postorder does not require any major description, even as much as we used to describe preorder, so what you do is whenever...

So, it is a recursive call, if you at the root and the root does not have, if the root does not have any children in this case, that is it is a leaf that you print the data item and return. Otherwise, you go to each of the children, you start up with the left most child and you do a postorder traversal of all its children and then move on to the right sibling and do a postorder traversal. So, there is a small error here, though I compiled it and everything this is the right.

(Refer Slide Time: 51:58)



```
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ ls tree+
tree-interface.h      tree-methods.o      treeevalExpr.c
tree-methods.c        tree.h               treeevalExpr
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ vi tree.h
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ vi tree-interface.h
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ ls
BigAddSub              PerfDLL              merge                tree-interface.h
BigAddSub.c            PerfDLL.c            merge.c              tree-methods.c
BinSchComp             balanced             queue-array.c         tree-methods.o
BinSchComp.c           balanced-paranthesis.c queue-array.o         tree.h
DLList-Interface.h     binaryTree-Interface.h queue-Interface.h     treeevalExpr.c
DLList.h               binaryTree-methods.c queue.h               treeevalExpr
DLListMethods.c         binaryTree-methods.o stack-array.c         week1-prog1.c
DLListMethods.o         binaryTree.h         stack-array.o         week1-prog2.c
List-Interface.h        bIneval              stack-Interface.h     week1-prog2.c
List.h                  bInevaluateExpression.c stack-list.c          week1-prog3.c
ListMethods.c           in-to-post           stack-list.o          week1-prog3.c
ListMethods.o           in-to-postfix.c      stack.h
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ vi tree-methods.c
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ gcc -c tree-methods.c
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ vi treeevalExp
```

So, that completes the implementation of all the tree methods, so let me just compile this, the compilation has succeeded and let us just look at the programming exercise. ((Refer

Time: 52:20)) So, essentially what we have return is we have return the expression and evolve it. Of course, our usage of the expression evaluator is very simple, so what we do is we first print the expression and the way you print the expression is it is a modification of the traversal techniques that we looked at. So, all that we do just is to just print a open parenthesis, first before we go and visit the descendant of a particular node.

(Refer Slide Time: 53:02)

```

}

/* A helper function to avoid few key strokes (Reusability) */
/* This function creates a Node and returns */
Node * createNode(char ch)
{
    Node *root = (Node *)malloc(sizeof(Node));
    root->data = ch;
    root->leftSibling = NULL;
    root->rightSibling = NULL;
    root->parent = NULL;
    root->leftChild = NULL;
    return root;
}

int main()
{
    Tree *t = createTree(); // note that we are using general trees

    // We will create the following tree
    //
    //      +
    //     / \
    //    *   *
    //   / \ / \
    //  2  - 3 2
    //   / \
    //  5  1

    Node *root = createNode('+'); // Create + as root;
    t->root = root;

    Node *new = createNode('*');
    insert(t, new, t->root); // insert * as child of root

```

So, let us look at the main function, the expression that we have encoded is this expression, so let us just see this expression the root node is plus, so now you can imagine that you can look at this expression, do a postorder traversal and do a traversal and then evaluate the expression. Do a postorder traversal and then evaluate the whole expression, so just populated the values here.

(Refer Slide Time: 53:40)

```
//
//
//      *
//     / \
//    2   3
//   / \ / \
//  5  1 2  2
//
Node *root = createNode('+');           // Create + as root;
t->root = root;

Node *new = createNode('*');
insert(t, new, t->root);                // insert * as child of root

new = createNode('2');
insert(t, new, t->root->leftChild);      // insert 2 as child of *
new = createNode('3');
insert(t, new, t->root->leftChild);      // insert 3 as child of '*'. Note that 2 is inserted first and then 3. This is because // insert() inserts every new node as the leftmost child of the parent
// Similarly insert other nodes
new = createNode('1');
insert(t, new, t->root);
new = createNode('-');
insert(t, new, t->root->leftChild);
new = createNode('1');
insert(t, new, t->root->leftChild->leftChild);
new = createNode('5');
insert(t, new, t->root->leftChild->leftChild);
new = createNode('2');
insert(t, new, t->root->leftChild);
printf("The value of the expression : ");
printExpr(t->root);
printf(" = %d\n", evalExpr(t->root));
}
"treeevalExpr.c" 100L, 2822C written
```

So, let us just look at the printed output.

(Refer Slide Time: 53:48)

```
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ ls tree*
tree-interface.h  tree-methods.o  treeevalExpr.c
tree-methods.c   tree.h          treeevalExpr.o
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ vi tree.h
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ vi tree-interface.h
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ ls
BigAddSub      PerfDLL      merge        tree-interface.h
BigAddSub.c    PerfDLL.c    merge.c      tree-methods.c
BinSchComp.c   balanced     queue-array.c tree-methods.o
BinSchComp.c   balanced-paranthesis.c queue-array.o tree.h
DLList-interface.h binaryTree-interface.h queue-interface.h treeevalExpr.c
DLList.h        binaryTree-methods.c queue.h       treeevalExpr.o
DLListMethods.c binaryTree-methods.o stack-array.c week1-prog1.c
DLListMethods.o binaryTree.h      stack-array.o week1-prog2.c
List-interface.h bineval         stack-interface.h week1-prog2.c
List.h          binevaluateExpression.c stack-list.c  week1-prog3.c
ListMethods.c   in-to-post    stack-list.o week1-prog3.c
ListMethods.o   in-to-postfix.c stack.h
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ vi tree-methods.c
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ gcc -c tree-methods.c
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ vi treeevalExpr.c
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ gcc treeevalExpr.c -o treeeval tree-methods.o
NSMarayanaswamy-MacBook-Air:code narayanaswamy$ ./treeeval
The value of the expression : ((2*(5-1))+(3*2)) = 14
NSMarayanaswamy-MacBook-Air:code narayanaswamy$
```

So, the executable that we create a treeeval and now I need to ensure that the tree methods dot o file is included with the compilation must complete, completed and let us now as you can see the expression is really this. It is the whole expression is parenthesis. It is 2 multiplied by 5 minus 1 that is the 5 minus 1 is a inner most expression, the expressions surrounding at 2 multiplied by 5 minus 1 plus, it start 2 and the whole thing forms the expression.

So, this is obviously a slightly modified version of printing it out in a certain format and then evaluating it using the postorder traversal technique. So, that brings to an end the first lecture on non linear data structures and we have really looked at the whole story about the implementation of trees. We will look at the motivation, we will look at all the context association with the tree, we will look at the different methods and we have seen an implementation of the methods.

What I have done this time is that I have rush through the C program, just to show you that indeed the implementation has done and the executable works and I show you a demo, this is to motivate you to go and try to do the implementation yourself by looking at just the video and the slides. This is definitely not going to be a programming exercise, this must be a self learning exercise that you must do.

So, when we meet next, we will look at binary trees which are a special kind of trees and we will look at the difference between binary trees and the trees that we have studied in this lecture and in the sub sequent lecture, we will look at some quick applications that bring us to an end this lecture.

Thank you very much.