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

Lecture – 15
Binary Tree ADT and traversals

Welcome to the second lecture on non linear data structures, and in the last lecture we talked about the tree data structure, and we looked at the abstract data type, the motivations for the tree data structure, the kind of the application that we have. And then we went through the whole implementation of the tree data structure, we also saw the code that was written for traversing the tree data structure and to evaluate some expressions.

So, in today's lecture, we are going to look at a special kind of a tree which is called a binary tree and you are going to see the binary tree carries forward, many properties of the tree abstract data type. And it also has additional interesting properties which are very useful when we want to represent data in a binary tree. We are going to therefore explore the binary tree abstract data type today, and really the methods that you see in binary tree abstract data type are not very different from the methods that you see in the tree abstract data type, except that there are some basic conventions which are very important, that is what we will be exposed. In particular for every node, if it has children nodes, then the binary tree has a special order among them, whereas the generic tree data structure does not mandate an order among the children. Though, the implementation will definitely an invariably have an order among the children in the tree abstract data type.

But, in the description of binary tree abstract data type, the children of a particular node, if well defined or ordered by the fact that they had the children and there is a special order among them. This is the very fundamental difference, we will study this today. As throughout this course many of the methods that we implement and specify are going to be recursive and we will also see that when we state certain traversal properties, we will actually stated using the power of recursion.

In other words, whenever we state a certain traversal approach on a binary tree or like

within a last lecture on the tree abstract data type, we just specify the behavior of the traversal at a particular node. In particular, we say when we mark the node as visited and what we will do it to it is children, how we will traverse the trees rooted, the sub trees rooted at the children of a particular node.

We will again do something similar, when we have traverse a binary tree, we will actually state the order in which the children of a particular node are visited and the order in which the sub trees rooted at the children are traverse. Therefore, we take recursion time and again and this is really a central theme in any data structures course that you must becomes comfortable with a usage of recursion, you must be able to understand recursive specifications.

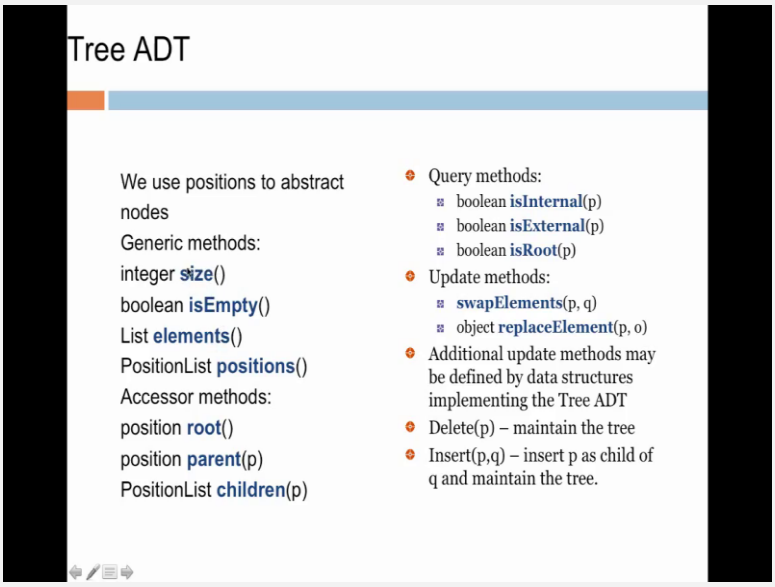
So, now let us go the slides, so the lecture pattern will more or less be similar to what you have been seeing so far. We will look at the slides, then we will look at the code and then we will have a quick short run and I would like you to draw your attention to fact that the description of the C program that I have written, I have reduce the time that I have spent on the descriptions, because by now you must be comfortable with more or less understanding, what the C program should be doing by understanding the slides itself.

Now, the other very important point that I will draw your attention to again is when you look at the C programs that I present to you, you will see that the C programs, the interface functions and the method are all broken into pieces of code which typically have a few statements maximum. You will see that many of them have one or two statements and some of them have four to five statements, so this is a kind of a Thumb rule.

If you are creating an abstract data type, then it is important for you to understand the abstract data type and understand it is methods in such a way that you will be able to implement each method using very few statements. So, this is very, very important thinking and really there is no specification as to what few is, but has a student programmer, I would say about most programming exercises is that you see either in a programming competition or in a course assignment, you should be able to write down

every method using not more than ten statements. So, that should typically be your goal. So, let us go to our slides and more on that as we go along.

(Refer Slide Time: 05:25)



We are going to look at the binary tree abstract data type, this is our lecture number 16 and since the description of the tree abstract data type, I am sure you recall most of it, there are generic methods which tell you the size of the tree, which means that tells you the number of nodes in the tree, whether the tree is empty and the most important thing, the reason I am repeating this part is to draw a distinction between these two.

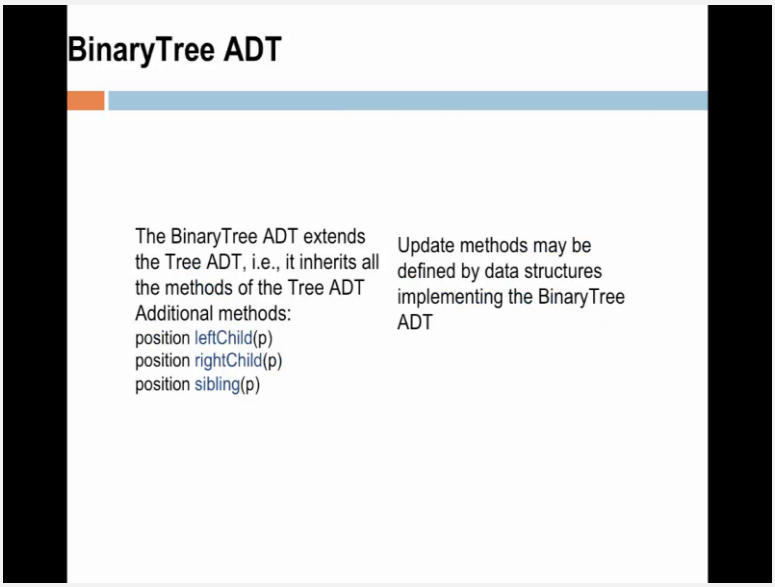
The elements method list all the elements which are in the tree, the positions method list all the positions or all the nodes in the tree. So, now off course the output of the positions method is not clear, all that you have to imagine is that the position method returns to you a list of pointers to individual positions in the tree or individual nodes in the tree. I repeat this, the position method will return a list, it will be a list of positions and every element in this list of positions will be a pointer to a node in the tree.

We do not really implement these in the C programs as we talked about, but all these are extremely easy to construct. Then, you have accessor methods which give you nodes or in other words, these are all positions parent of p, returns a pointer to the parent of the

node p, if it is well defined, children of p returns a position list which each element in the list is the pointer to the children p. The query methods are properties of every node, whether it is an internal node or an external node, the update methods are the once that deal with the values which we stored at different nodes in the tree.

Therefore, the most important thing that I am repeating in this module is that there are two kinds of objects that you manipulate in a tree abstract data type as a programmer. One is the structural aspect that is you manipulate the pointers to maintain the tree like structure. Second, you manipulate the values which are stored in the nodes; therefore, it is very important to keep this distinction in mind. Therefore, tree is a structural property and the data which are organized in a tree are the ones that you want to manipulate. And those that you want to access and update and modify, query and so on and so forth, so this is the story of the tree abstract data type.

(Refer Slide Time: 07:50)



What is the binary tree abstract data type? First of all, the binary tree abstract data type has all the methods and the properties associated with a tree abstract data type. Apart from this, it has additional methods, position methods which now tell you the left child and the right child. Therefore, if p is a node, the left child is a pointer to a special child which is called the left child of p and the right child is a pointer to a special child which

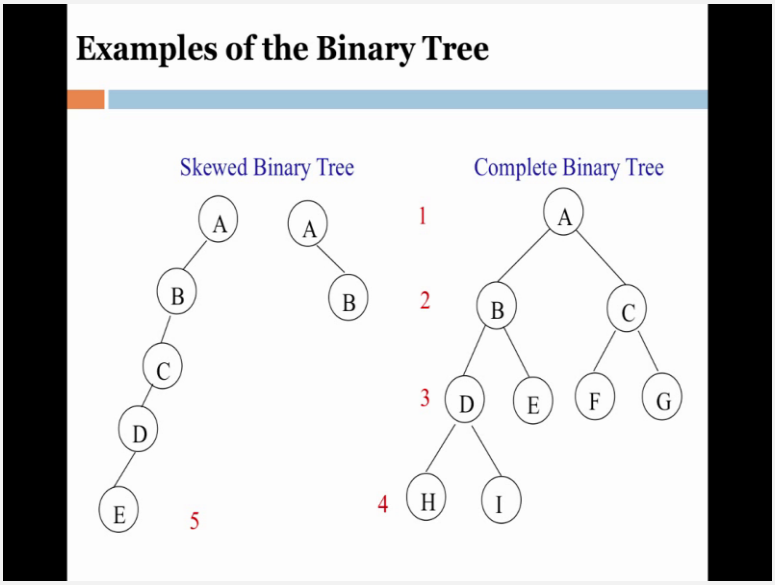
is called the right child of p.

As you can see now that by the definition of these additional methods as I have just specified them, every node has at most two children, such a tree is a binary tree. Therefore, the property of the binary tree abstract data type is that it has all the methods and the data items that associated with the tree ADT plus it has two additional methods, one for the left child of p and one for the right child of p and it is possible that either exactly one of the two children are well defined or neither is well defined.

Another method that we have is a sibling of p and if you notice what I said, I said the sibling of p. Therefore, recall that in the tree abstract data type, the sibling would be a listing of nodes it would be a position list. On the other hand, very importantly the return value of this particular method is a pointer to one another node, in other word it is a position. Therefore, let us just recall the structure of the binary tree, it is a tree first of all, it has all the data and method associated with the tree, it has three additional methods which essentially place further restrictions on the structure of the tree.

So, every node has at most one left child and at most right child and every node has at most one sibling. The update methods may be define by the different data structures, we will actually see in the next set of lectures, something called a binary search tree and we will look at the heap data structure. And both of these will have their own update method implementations. In particular, we will be able to update the values in the different nodes, we will be able to delete certain values, we will be able to delete certain nodes and so on and so forth.

(Refer Slide Time: 10:25)



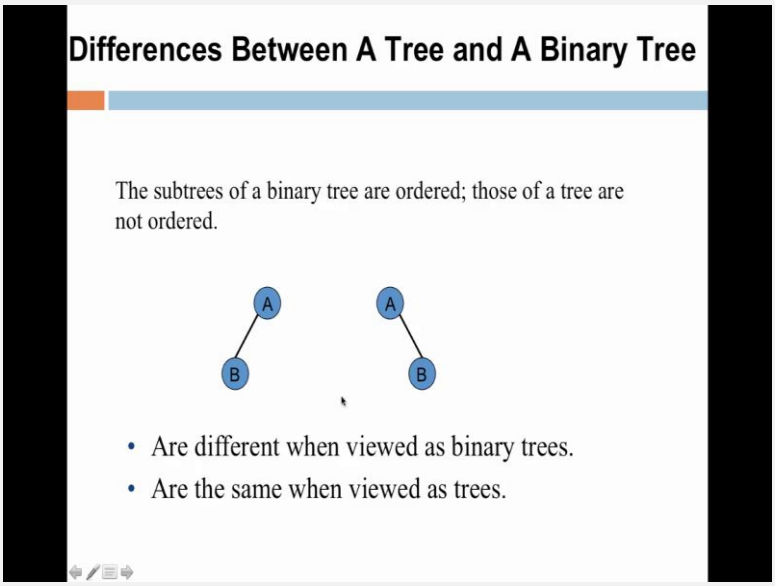
Let us look at some examples of binary tree, the best way to understand the distinction between different concepts is to look at a few examples, let us look at a skewed binary tree, it is called as skewed binary tree, because if you notice for every node here, one of the children is missing. So, this is also a binary tree, A is the root of the binary tree, B is it left child, C is the left child, D is the left child of C, E is the left child of D.

In this case, a different skewed binary tree, A is the parent of B and B is the right child of A. On the other hand, this is a complete binary tree, a complete binary tree is one and which every internal node that is every non external node has exactly two children. I repeat, a complete binary tree is one in which every internal node otherwise a non external node observe that E, F and G are external nodes, H, I are external nodes, they have zero children whereas D, B, A and C have exactly two children. This is the distinction between a skewed binary tree and a complete binary tree.

Now, you can ask yourself a very important question, what is the depth of a complete binary tree? And if you have two paths, one from the root to this leaf and one from the root to another leaf, what can the depth of the two nodes, two leaf nodes. How much can they differ by? This is an interesting question and you can ask yourself this question, we will visit this in a couple of lectures from now. Let me post the question clearly, the

depth of every node is a well defined concept from the previous lecture. In a complete binary tree, what can be the largest value of the difference between the depth of two leaf nodes.

(Refer Slide Time: 12:35)



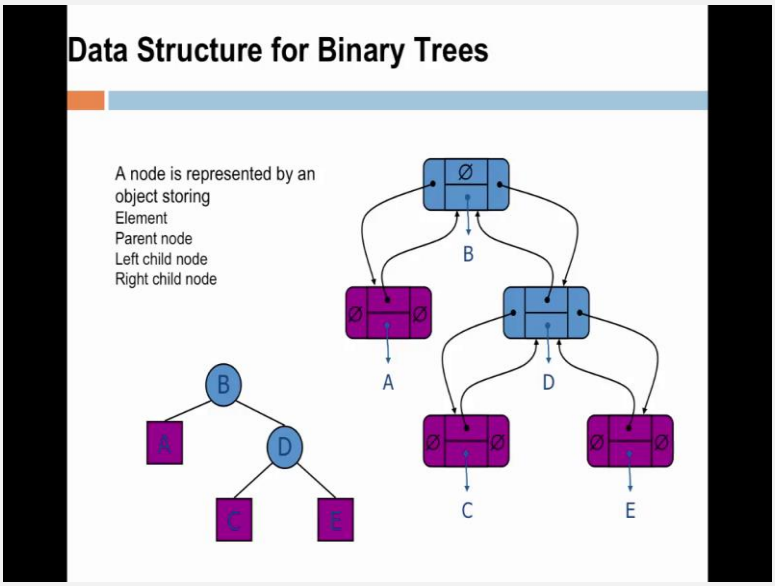
Now, comes to the concept of the fundamental difference between an element of the tree abstract data type and an element of the binary tree abstract data type. The sub trees of a binary tree are ordered. In particular, we say that B is a left child of A. In this example, I say that B is a right child of A, now these two trees that we have drawn are two different binary trees. In this binary tree, B is a left child of A and in this binary tree B is a right child of A.

On the other hand, when we view this as trees, both of them are the same, B is just a child of A, B is just a child of A. Therefore, the sibling relationship from the concept of ADT really does not exists, similarly the child relationship being the left child and the right does not exists when you define a binary tree, it does not exists when you define an arbitrary tree. When you implement, there will be the first child, the second child, the third child and so on, because we keep track of the left most child right sibling.

Recall the left most child right sibling representation of a tree, so in the implementation

there is a well defined ordering of the children. But, in the specification of the abstract data type we do not have any such mandate to have an ordering of children, whereas in a binary tree, we definitely have an ordering of the at most two children of any node.

(Refer Slide Time: 14:08)

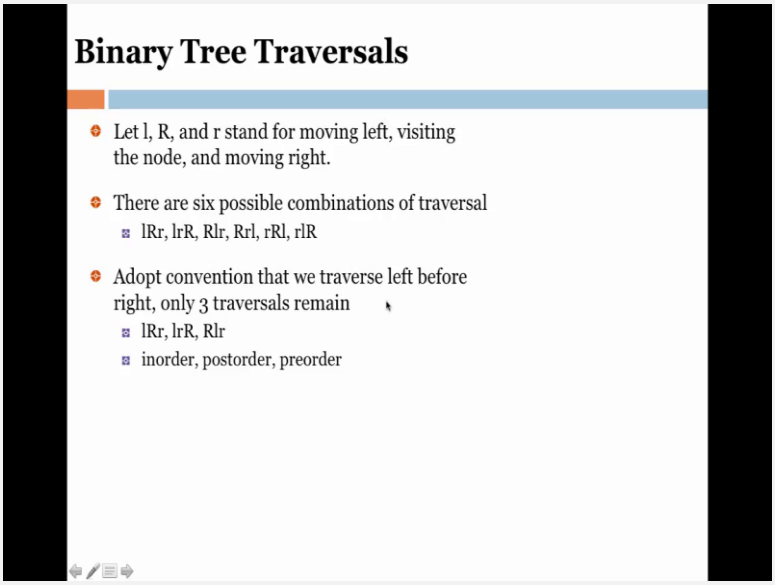


Here is the data structure for the binary tree, it use an implementation in memory view. This on the other hand is an abstract view in a mind of a programmer or a designer, this is the in memory view in the mind of a programmer or the person, who is coding the whole data structure or who is writing the program for setting of the data structure. Now, what we have is a node, the nodes are of two colors as you can see, the violet color or the pink color nodes are all external node, otherwise the leaf nodes, where both the left pointer and the right pointer are null pointers.

There is value feel which has a value A and for the sack of efficiency we also store a parent pointer. That parent pointer for the root node is that the parent pointer points to null. As usual we have an element that pointer to the parent node, a left child node and a right child node. So, now you can ask yourself the question, what is the sibling of D? The sibling of the node containing D is the node containing A. In particular, you can say that the left sibling of the node containing D is the left sibling of the node containing A. Similarly, the right sibling of the node containing A is the right sibling of a node

containing D.

(Refer Slide Time: 15:30)



Now, let us move on to binary tree traversals and as you can see I am using the power of recursion to actually state this in a very crisp fashion. Let us assume that l, capital R and small r stands for moving left, visiting a node and moving right. Therefore, for the recursive specification of the different kinds of traversals, there are six possible rules that could be followed. You follow, you first explore the left sub tree, then you mark the current node as visited, then you visit the right sub tree.

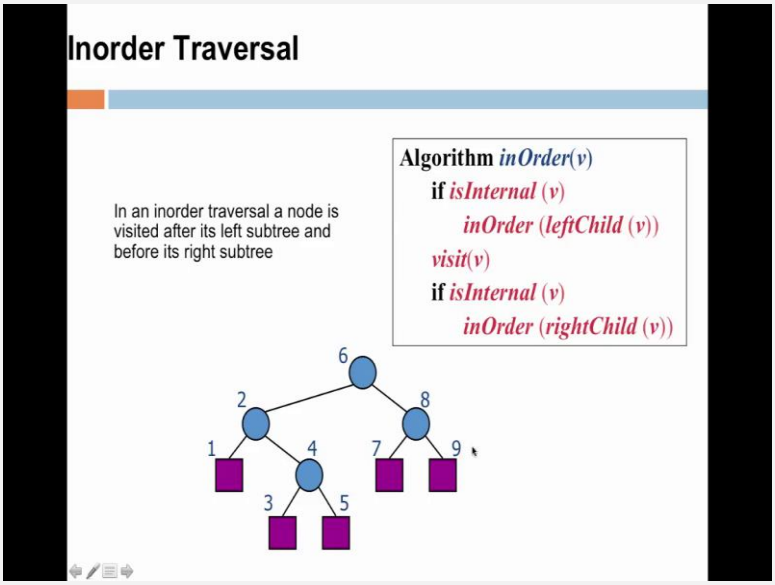
Or you visit the left sub tree, you traversal the left sub tree, then you traversal the right sub tree, then mark the node as visited. Now, the third possibility is mark the node as visited, then you traverse the left sub tree, then you traverse the right sub tree. Observe that the traversal is specified recursively the word traversal is used recursively. Therefore, there are six such possibilities as clear, there are three factorial, there are three possibilities left, right and capital R, that is to visit the node and these can be done in six ways which is three factorial, three multiplied and two multiplied, then one possible ways.

If you adopt the convention that we traverse the left before the right, only three traversals

remain. What are the three traversals? First you traverse the left sub tree, then you traverse, then you mark the current node as visited, then you traverse the right sub tree. This is called inorder traversal. If you traverse the left sub tree, then the right sub tree and then mark the node as visited, then this is called postorder traversal.

If you traverse the, if you mark the node as visited, then you traverse the left sub tree, then the right sub tree, then we have the preorder traversal. Observe that when we talked about traversals in the tree abstract data type, we really did not talked about the inorder traversal and on the other hand we did talked about the post order traversal.

(Refer Slide Time: 17:37)

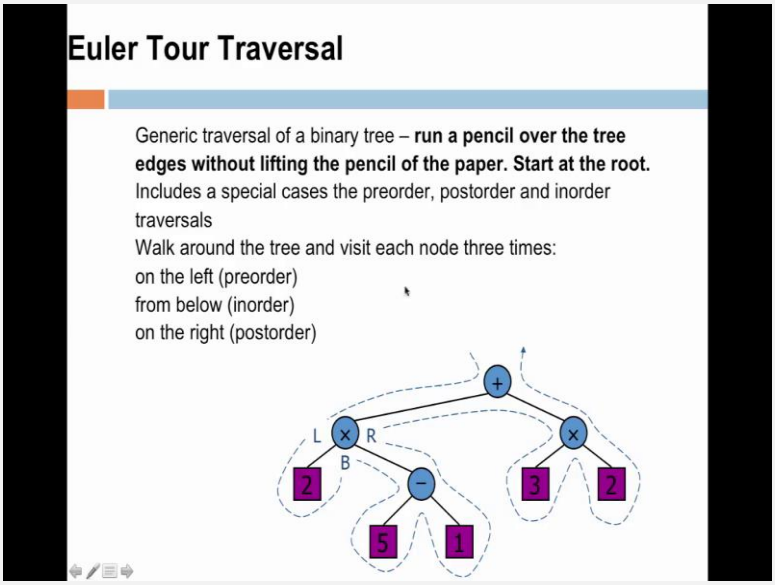


So, let us just look at the code for the inorder traversal and this is what it will be more or less implemented in the C program. The numbering associated with the nodes in the tree tell you the order in which they are printed out by the inorder traversal. So, what is the rule for the inorder traversal? You traverse, you start at the root, you traverse the left sub tree first, then you mark this as visited, then you traverse the right sub tree that is exactly what it says.

If the node is internal, then you do an inorder traversal of left child of v, you visit v and if it is an internal, then you perform an inorder traversal of the right child of v, I think this

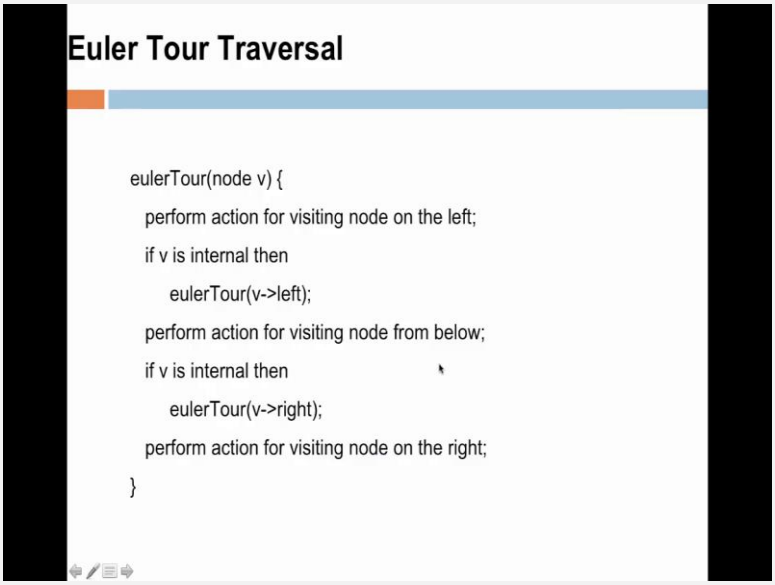
if is internal, it does not required here, but it does not matter, the code is indeed correct. So, let us just look at the order in which the nodes will be marked as visited, the first node to be marked as visited will be 1, then 2 gets marked as visited, then 3 gets marked as visited, then 4, then 5, then it is 6, then the right sub tree of the right child of, the sub tree rooted with the right child of this node, the root node and that you can see will be marked as 7, 8 and 9 respectively. This is the inorder traversal.

(Refer Slide Time: 19:02)



So, let us look at one more traversal and the code for this follows in the next slide, we have not yet implemented this to show you in the C program, but the picture is very clear. It is a nice traversal of the binary tree which is very interesting, you visualize the tree with a root and you run a pencil over the tree edges starting at the root, that is you start from here and you run a pencil and visit every edge without lifting the pencil from the paper. As you can see the, mouse pointer is moving exactly the way the traversal is supposed to happen. So, now you will see that every edge is visited twice.

(Refer Slide Time: 19:59)



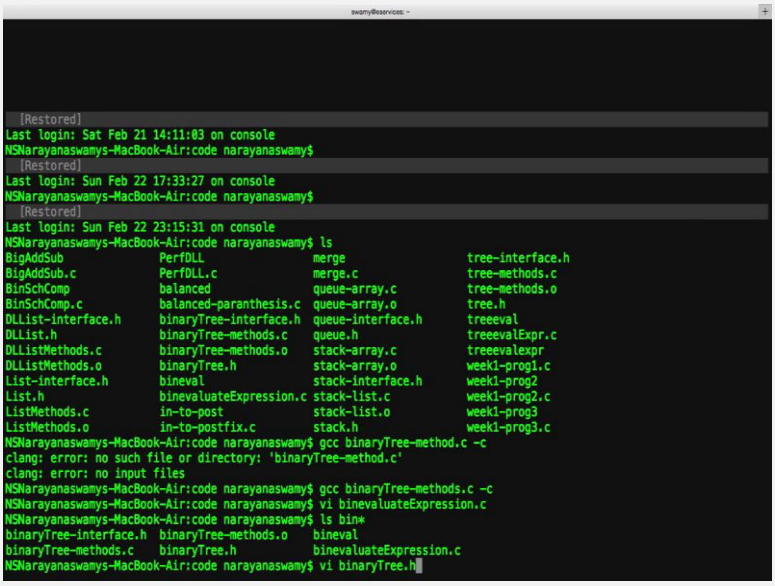
Let us just looked at a small function description, this is called the Euler tour traversal, so perform action for visiting node to the left and if v is internal, then you perform a Euler tour recursively at the left node. Then, you perform the action for visiting the node from below, then if it is internal, you move to the right and you perform the action for visiting the node to the right and this will be done recursively. So, let us just do this again.

So, you visit 2, then you come back to the root, then you go to the hand side and visit this. So, one can also visualize this work has been done by the inorder traversal of the tree that we just looked at. So, what we get just now was to look at the binary tree abstract data type and you would have observe that the conceptual differences between the binary tree abstract data type and the tree abstract data type is in the factor that the binary tree, the children are at most two in number and they are ordered and as a sibling relationship every node has at most one sibling, it has at most one left sibling or at most one right sibling.

And we also saw that one can visualize a new traversal of a binary tree which is an inorder traversal and from the presentation we would have also notice that the inorder traversal also gives you an idea of the order in which expressions are evaluated. So, if

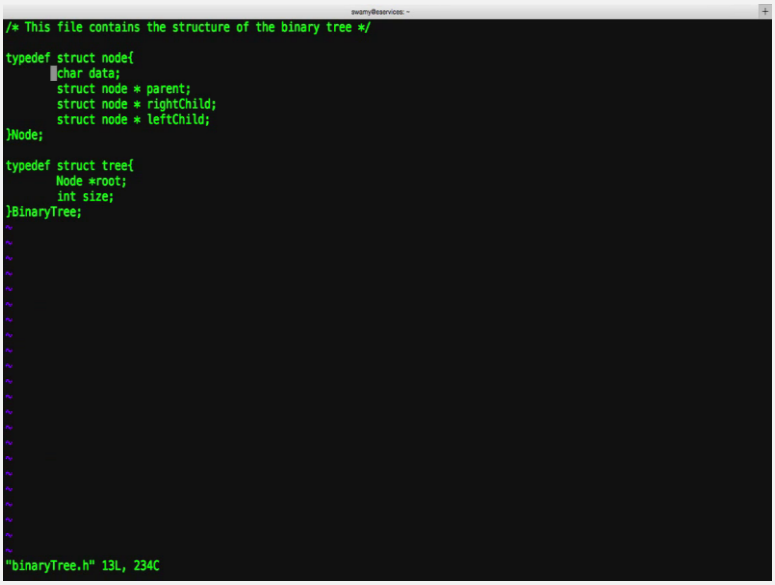
your tree was an expression tree, as we will see in the next lecture the order in which the tree is evaluated is essentially using the inorder traversal.

(Refer Slide Time: 22:05)



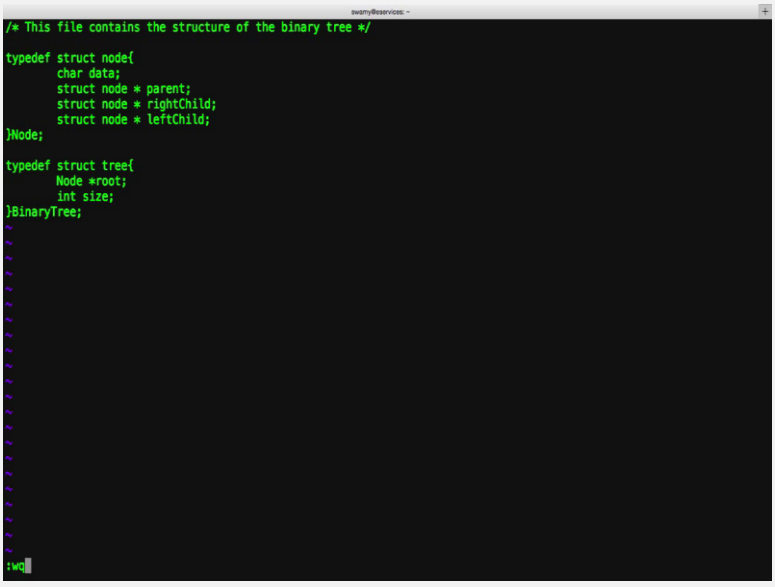
Let us now move to look at the programs that we have written and quickly go through the programs, because what you see in the C programs now is exactly what we saw in the slides, except that we can compile them and execute them. So, as you can see I have written four pieces of code two dot h files and dot c files. So, binary tree dot h tells you the corresponding basic data type, the ADT structure, the node structure for the binary tree and then we have the binary tree methods and these methods are in implementation of the ADT methods associated with a binary tree and then the interface functions are what are included in the C file and the binary tree methods is compiled to give a dot o file. So, let us just quickly go through…

(Refer Slide Time: 23:11)



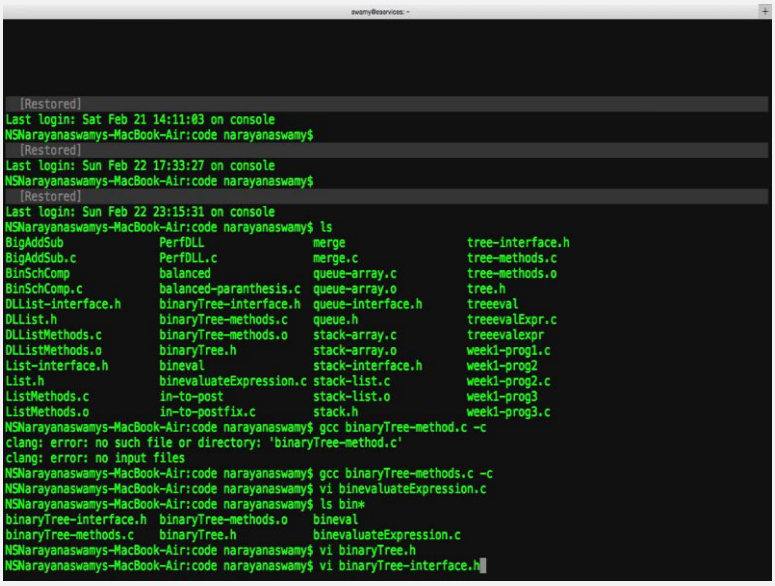
So, here is the node in the tree, you have a data item, you have a pointer to the parent, you have a pointer to the right child and you have a pointer to the left child and this is called a node. Now, we also have a type called the tree, I call it a binary tree. It is comprised of a root node and it has a data item which is the size. So, therefore the tree itself is just the pointer to the root node of the tree.

(Refer Slide Time: 23:47)



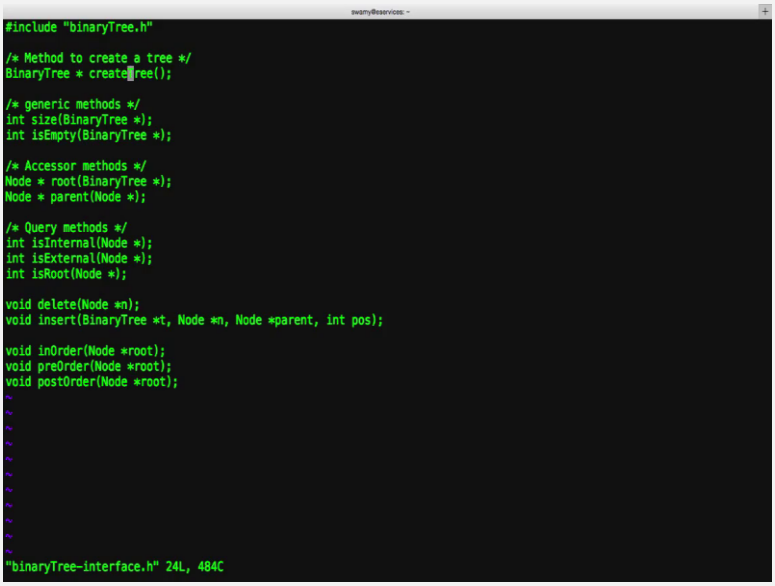
So, this is the structure of the binary tree, so this essentially tells you what the data looks like.

(Refer Slide Time: 23:54)



Let us look at the interface function.

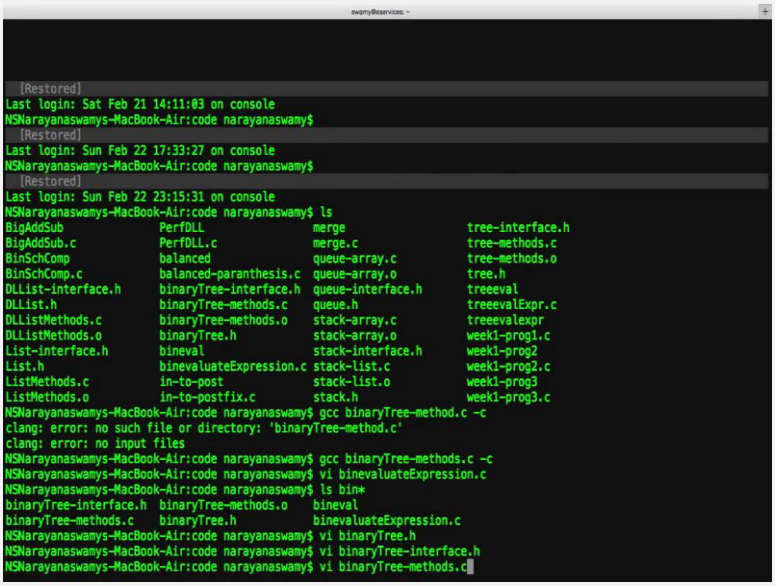
(Refer Slide Time: 24:01)



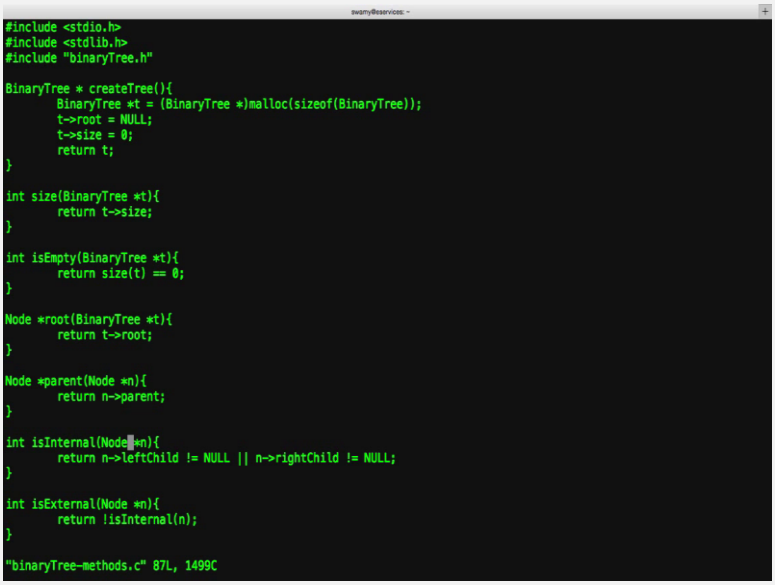
So, you include binary tree dot h and this is a method to create a tree or initiates a tree, we just creates one node and I am basically initializes all the pointers appropriately. These functions are the generic methods that we have seen the accessor methods which gives you pointers to different nodes in the tree, the pointer to the parent, pointer to the root and these are query methods, structural query methods to ask whether a node is an internal node or an external node, these are Boolean queries as you can see and here is a query asked to whether a node is a root of a tree or not.

This is to check if the parent pointer is null, here we delete nodes, you have been insert into the tree t and node n as a parent, with this as a parent and the inorder, preorder and postorder traversal also implemented, given a pointer to the root of the particular tree, these are the interface methods.

(Refer Slide Time: 25:15)



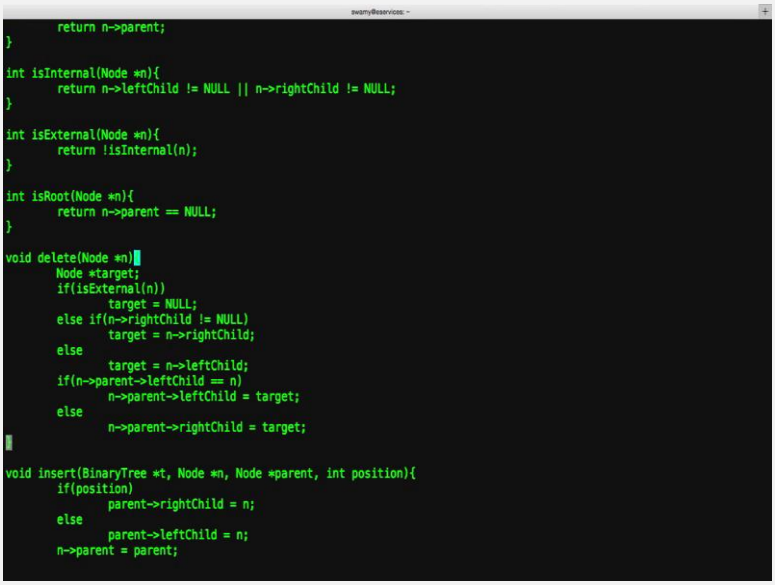
(Refer Slide Time: 25:21)



Let us look at the… Let us quickly run through the constructor function of the createTree function, the createTree function creates a binary tree, it says that the root value points to null, the tree is empty there is nothing else, the tree is empty by certain t pointer at size is equal to 0. The size function returns a value of t pointer of size, the tree is empty when there are no nodes inside it, when the size of t is equal to 0. As you can see that each of these functions are a very important and they are just a single statement.

To get the root of a tree which is return a pointer to the root and to get a pointer to the parent of a particular node, you return the node pointer to parent. To check if a node is null, you will check if either the left child is not null or the right child is not null. Observe that there is a single logical expression takes care of this.

(Refer Slide Time: 26:26)



And a node is external, if it is not internal this is nicely captured in this statement. If something is a root, then it is parent is null as I said earlier, now let us look at the delete function. So, let us look at this function carefully and then move on, because this is the most crucial new thing in an implementation from the last lecture. So, you want to delete the node n, so to delete the node n, what we do is we have to first keep track of the parent of this particular node and look at the child and look at the children of this particular node and connect it appropriately.

So, let us just see this, if the node is external then there is no work to be done, target is null, then the delete function will deleted. If the node has a right child, then the target is taken to be the right child of the node, otherwise it is taken to be the left child of the node. And observe that implicitly, if a node has both, it is right child and left child, we did not specify completely what the deletion as to do, we will see this in a subsequent implementation of binary search trees.

But, in this code we always delete a node, if it is either leaf or has exactly a left child or has exactly a right child. The picture is quite clear, if it is a leaf no work needs to be done, we can just be deleted and if it is really the left child of it is parent, then the left child to the parent now points to the left child of the or this target, depending on whether

it is a left child or a right child. The same thing here, if the node being deleted is a right child of it is parent, then the right child of the parent now becomes a target which is the single child of the node that you have deleted.

So, let us just go through this again, the convention here is that we will delete only nodes which have either zero or one children, zero children or one child and if it is zero, no work needs to be done. We need to program this more carefully, but just to show you an implementation which has been quickly done and if it has exactly one child, then that single child is stored in this variable target, it is a pointer and the left child or the right child to the parent is then set to target. This way the node n is no longer in the tree.

(Refer Slide Time: 29:47)

```
int isRoot(Node *n){
        return n->parent == NULL;
}

void delete(Node *n){
        Node *target;
        if(isExternal(n))
                target = NULL;
        else if(n->rightChild != NULL)
                target = n->rightChild;
        else
                target = n->leftChild;
        if(n->parent->leftChild == n)
                n->parent->leftChild = target;
        else
                n->parent->rightChild = target;
}

void insert(BinaryTree *t, Node *n, Node *parent, int position)
        if(position)
                parent->rightChild = n;
        else
                parent->leftChild = n;
        n->parent = parent;
        t->size = t->size + 1;


void inOrder(Node *root){
        if(root == NULL)
                return;
        if(isInternal(root))
                inOrder(root->leftChild);
        printf("%c ", root->data);
        if(isInternal(root))
                inOrder(root->rightChild);
```

Insert again is fairly simple, so what we do is, we insert the node based on whether it is the left child or the right child of the parent. So, the position value tells you whether the node n has to be inserted has a left child or a right child of the parent and here the assumption is that the programmer text care of carefully inserting it by giving the appropriate values in a position.

So, depending on position and if a position is 1, then you insert parent pointer to the right child as n and otherwise we insert parent pointer or left child to be the node n and the

parent of the node n is to be is given to be the parent and the size of the tree is increased. Observe that the only place a node gets inserted is at a leaf node. Of course, this is not checked here, this is the job of the insertion function. Whichever function is doing the insertion has to do this carefully.

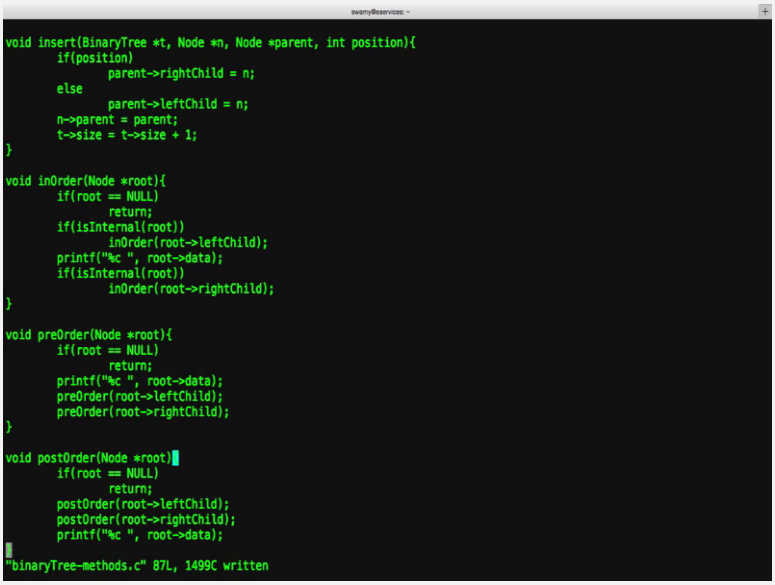
It has to find the leaf and ensure that node n is the appropriate leaf or leaf or a node with exactly one child and insert, find the parent which has either zero children or exactly one child and inserted based on the value of position. Let me repeat this, so this insert function inserted into a tree, inserts n into a tree as a child of a node called parent and n will be inserted based on the value of position. The value of position will be either 1 or 0 and if it is 1, n is made the right child of parent, if n 0 it is made the left child of parent and the size of the tree is incremented appropriately. It is the responsibility or the calling function to check that this implementation is used meaningfully.

(Refer Slide Time: 31:53)

```
                target = NULL;
        else if(n->rightChild != NULL)
                target = n->rightChild;
        else
                target = n->leftChild;
        if(n->parent->leftChild == n)
                n->parent->leftChild = target;
        else
                n->parent->rightChild = target;
}

void insert(BinaryTree *t, Node *n, Node *parent, int position){
        if(position)
                parent->rightChild = n;
        else
                parent->leftChild = n;
        n->parent = parent;
        t->size = t->size + 1;
}

void inOrder(Node *root){
        if(root == NULL)
                return;
        if(isInternal(root))
                inOrder(root->leftChild);
        printf("%c ", root->data);
        if(isInternal(root))
                inOrder(root->rightChild);
}

void preOrder(Node *root){
        if(root == NULL)
                return;
        printf("%c ", root->data);
        preOrder(root->leftChild);
```

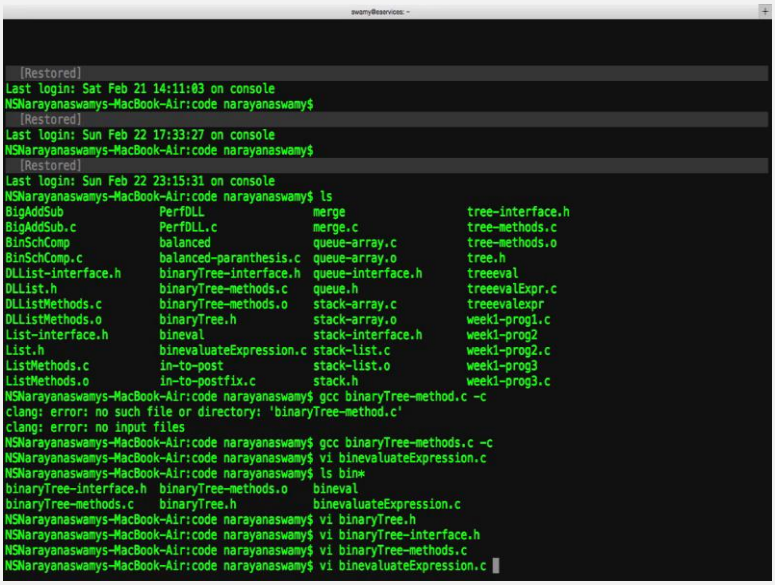
Here at the traversals which are fairly straight forward, the inorder traversal as you can see is the listing where we just print out the value at different nodes in the tree.

(Refer Slide Time: 32:06)



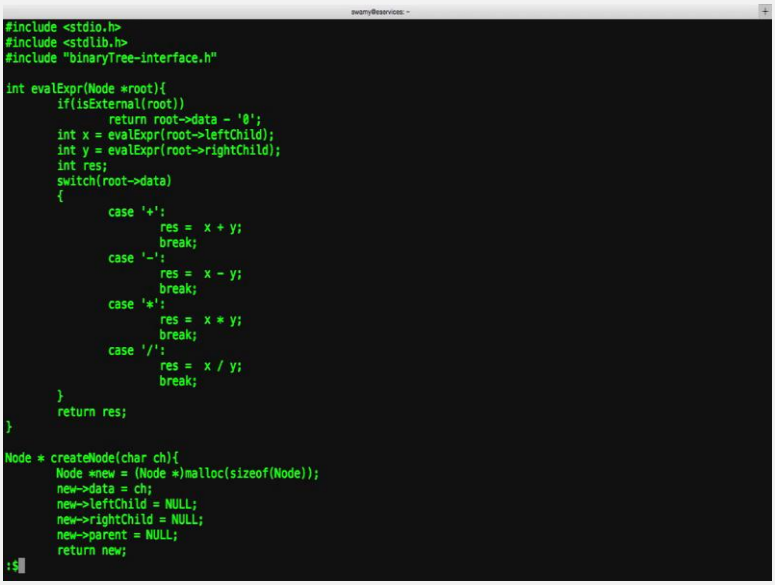
So, as you can see we traverse a left child, then print the root, then traverse the right child, whereas in preorder we print the root, traverse the left child then traverse the right child. In postorder, we traverse the left child then traverse the right child, then we print the root. This completes the implementation of all the methods that is specified with the binary tree.

(Refer Slide Time: 32:30)

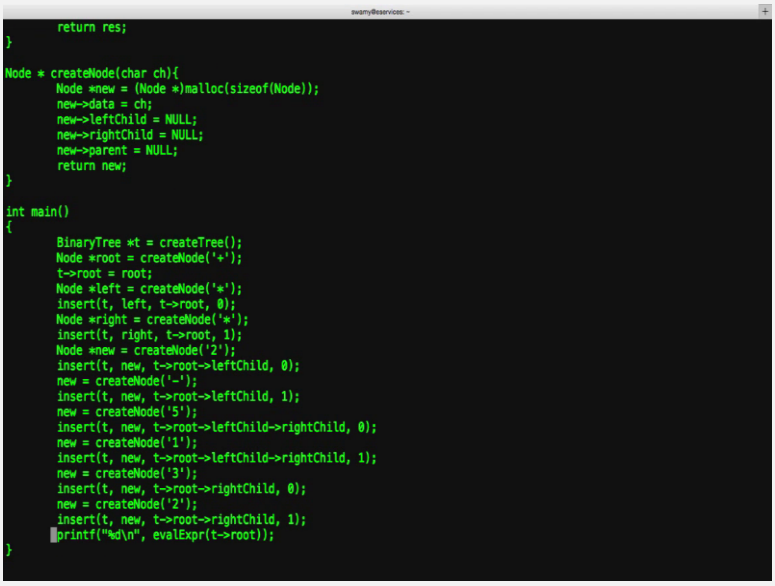


Let us look at an example exercise.

(Refer slide Time: 32:36)



(Refer Slide Time: 32:43)



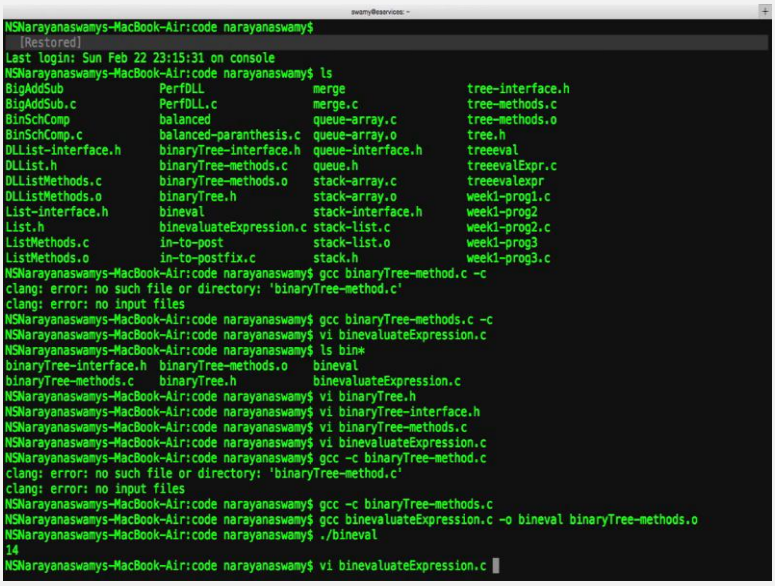
So, the exercise here is to create a tree and populate the tree with an arithmetic expression. So, the expression as use you will see is the following. The root is a plus, the left child is a star and to the left of the left child we insert the value zero and at the right

of the root we create the star. So, as you can see at the left of the node we have the multiplication operator and at the right of the root we have the multiplication operator again and then we populate values in the tree and we have essentially add code as a simple arithmetic expression.

What you can definitely do is to write a small function which will read from the input an arithmetic expression and construct this tree. The more important thing is the evaluation, let us just look at this, the evaluate expression is a recursive function, you will see that it evaluates the expression at the left child, then evaluate the expression at the right child, takes the two value and depending on the operator which is present at the root, then it performs the appropriate arithmetic operation returns it to the calling function.

So, it is a recursive function which evaluates the expression in the order in which we evaluate the arithmetic expressions which are given in the normal inorder parenthesized form. So, you evaluate the left expression first, then evaluate the right expression and then add it. So, this is exactly the evaluation that we do by hand and we look at an arithmetic expression. In this exercise, we assume the arithmetic expression is stored in a tree to which we have access via, the root pointer and then we evaluate the expression. As you can see, the evaluation is a postorder expression.

(Refer Slide Time: 35:02)



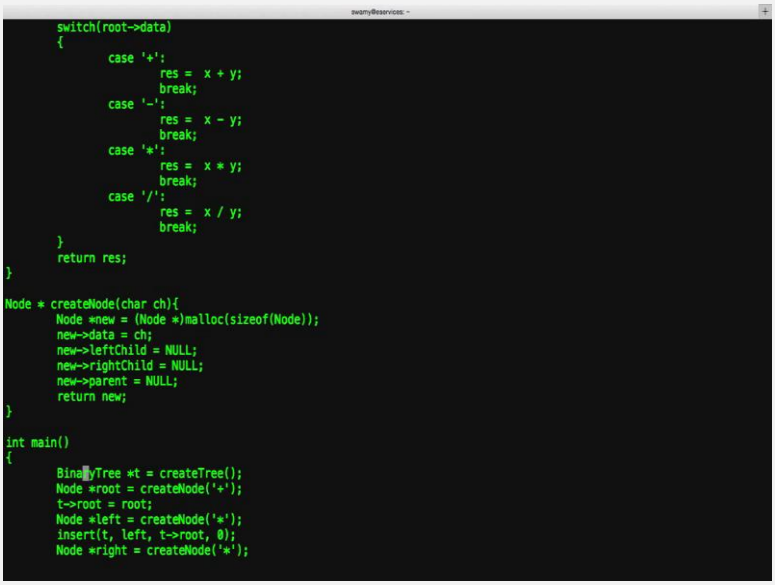
So, let us just compile all these things, so this compiles the file of binary tree methods, then we compile the program which is a TA program making use of all these functions, now use the object file that we have created. Now, the compilation is done and the executable is then evolved and the value is 14 for the expression that is ((Refer Time: 36:02)) coded. The value really does not a matter, let me just take you back to the…

(Refer Slide Time: 36:12)

```
#include <stdio.h>
#include <stdlib.h>
#include "binaryTree-interface.h"

int evalExpr(Node *root){
        if(isExternal(root))
                return root->data - '0';
        int x = evalExpr(root->leftChild);
        int y = evalExpr(root->rightChild);
        int res;
        switch(root->data)
        {
                case '+':
                        res =  x + y;
                        break;
                case '-':
                        res =  x - y;
                        break;
                case '*':
                        res =  x * y;
                        break;
                case '/':
                        res =  x / y;
                        break;
        }
        return res;
}

Node * createNode(char ch){
        Node *new = (Node *)malloc(sizeof(Node));
        new->data = ch;
        new->leftChild = NULL;
        new->rightChild = NULL;
        new->parent = NULL;
        return new;
"binevaluateExpression.c" 60L, 1274C
```

If you look at the code, we have used almost all the functions which we have written in the interface file. We have checked, if the node is an internal node or an external node, then we have access to the data items left child, right child and parent.

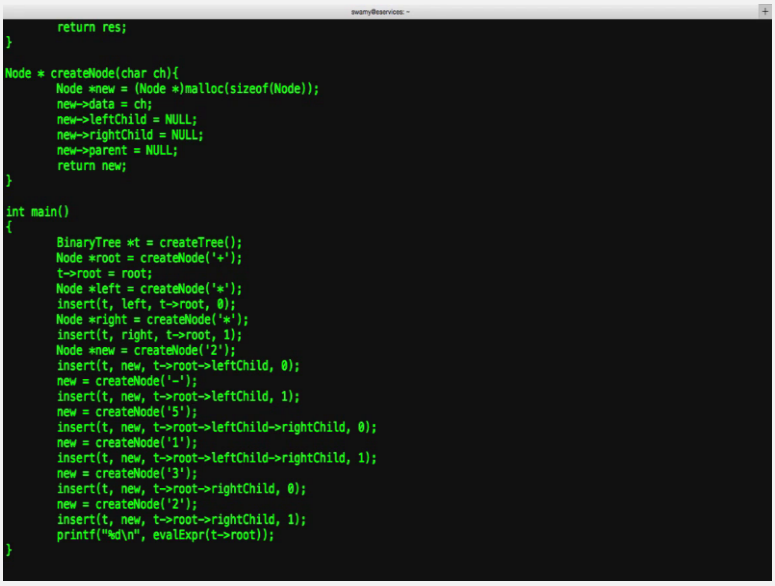
(Refer Slide Time: 36:27)



```
        switch(root->data)
        {
                case '+':
                        res =  x + y;
                        break;
                case '-':
                        res =  x - y;
                        break;
                case '*':
                        res =  x * y;
                        break;
                case '/':
                        res =  x / y;
                        break;
        }
        return res;
}

Node * createNode(char ch){
        Node *new = (Node *)malloc(sizeof(Node));
        new->data = ch;
        new->leftChild = NULL;
        new->rightChild = NULL;
        new->parent = NULL;
        return new;
}

int main()
{
        BinaryTree *t = createTree();
        Node *root = createNode('+');
        t->root = root;
        Node *left = createNode('*');
        insert(t, left, t->root, 0);
        Node *right = createNode('*');
```

Then, we use a create tree method which is a method to create a tree, it initiates a tree.

(Refer Slide Time: 36:38)



```
        return res;
}

Node * createNode(char ch){
        Node *new = (Node *)malloc(sizeof(Node));
        new->data = ch;
        new->leftChild = NULL;
        new->rightChild = NULL;
        new->parent = NULL;
        return new;
}

int main()
{
        BinaryTree *t = createTree();
        Node *root = createNode('+');
        t->root = root;
        Node *left = createNode('*');
        insert(t, left, t->root, 0);
        Node *right = createNode('*');
        insert(t, right, t->root, 1);
        Node *new = createNode('2');
        insert(t, new, t->root->leftChild, 0);
        new = createNode('-');
        insert(t, new, t->root->leftChild, 1);
        new = createNode('5');
        insert(t, new, t->root->leftChild->rightChild, 0);
        new = createNode('1');
        insert(t, new, t->root->leftChild->rightChild, 1);
        new = createNode('3');
        insert(t, new, t->root->rightChild, 0);
        new = createNode('2');
        insert(t, new, t->root->rightChild, 1);
        printf("%d\n", evalExpr(t->root));
}
```

And we use a insert function which inserts a node into the tree and finely we evaluate the expression by traversing the tree carefully and using the methods that we have implemented along with a binary tree ADT. So, that brings us to the end of this lecture on

the binary tree abstract data type, as you can see we have now a clear understanding of the distinction between a tree and a binary tree.

And this distinction is very important, because a binary tree has some very important order information as part of the abstract data type itself and this is extremely important to understand binary trees are used to store arithmetic expressions and binary trees are used to implement the board mass rules that we studied in schools, the rules to evaluate arithmetic expressions in an uniform and consistence fashion. In the next lecture when we meet, we will use all these methods that we have seen so far, and write few quick programs for expression, evaluation and other such exercises that brings to an end today's lecture.

Thank you.