Data Structures

Lecture No. 25

Reading Material

Data Structures and Algorithm Analysis in C++ Chapter. 4, 10
4.2.2, 10.1.2

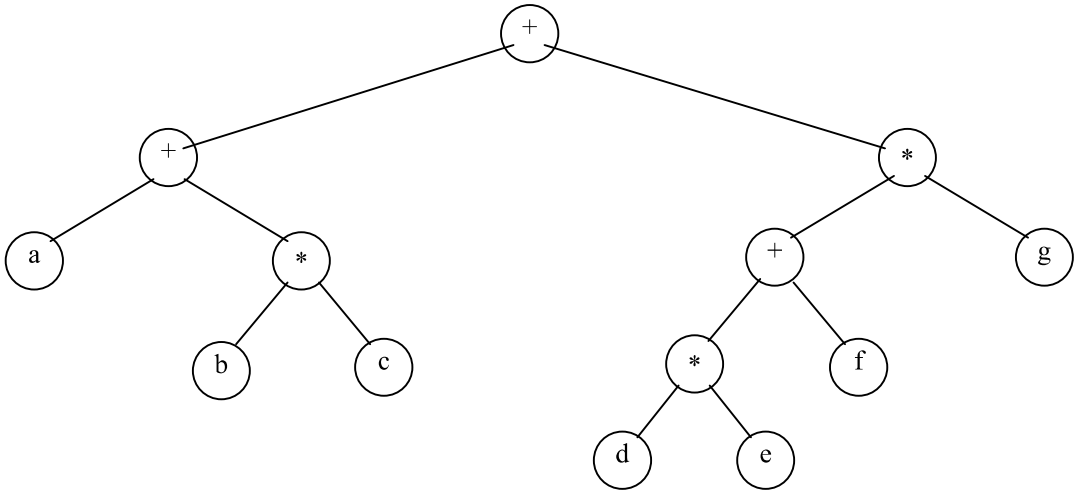
Summary

- Expression tree
- Huffman Encoding

Expression tree

We discussed the concept of expression trees in detail in the previous lecture. Trees are used in many other ways in the computer science. Compilers and database are two major examples in this regard. In case of compilers, when the languages are translated into machine language, tree-like structures are used. We have also seen an example of expression tree comprising the mathematical expression. Let's have more discussion

on the expression trees. We will see what are the benefits of expression trees and how can we build an expression tree. Following is the figure of an expression tree.

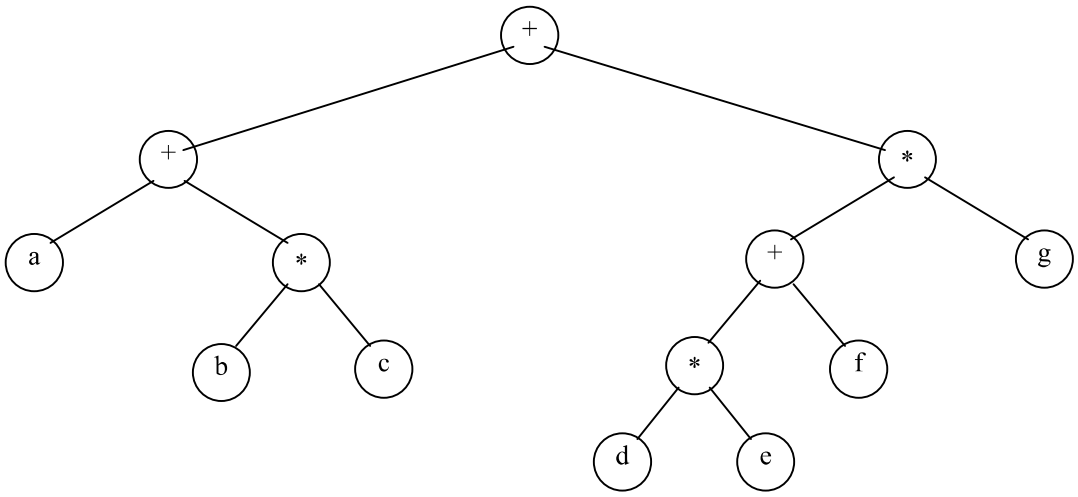


In the above tree, the expression on the left side is *a + b * c* while on the right side, we have *d * e + f * g*. If you look at the figure, it becomes evident that the inner nodes contain operators while leaf nodes have operands. We know that there are two types of nodes in the tree i.e. inner nodes and leaf nodes. The leaf nodes are such nodes which have left and right subtrees as null. You will find these at the bottom level of the tree. The leaf nodes are connected with the inner nodes. So in trees, we have some inner nodes and some leaf nodes.

In the above diagram, all the inner nodes (the nodes which have either left or right child or both) have operators. In this case, we have + or * as operators. Whereas leaf nodes contain operands only i.e. *a, b, c, d, e, f, g*. This tree is binary as the operators are binary. We have discussed the evaluation of postfix and infix expressions and have seen that the binary operators need two operands. In the infix expressions, one operand is on the left side of the operator and the other is on the right side. Suppose, if we have + operator, it will be written as *2 + 4*. However, in case of multiplication, we will write as *5*6*. We may have unary operators like negation (-) or in Boolean expression we have NOT. In this example, there are all the binary operators. Therefore, this tree is a binary tree. This is not the Binary Search Tree. In BST, the values on the left side of the nodes are smaller and the values on the right side are greater than the node. Therefore, this is not a BST. Here we have an expression tree with no sorting process involved.

This is not necessary that expression tree is always binary tree. Suppose we have a unary operator like negation. In this case, we have a node which has (-) in it and there is only one leaf node under it. It means just negate that operand.

Let's talk about the traversal of the expression tree. The inorder traversal may be executed here.



Inorder traversal yields: a+b*c+d*e+f*g

We use the inorder routine and give the root of this tree. The inorder traversal will be *a+b*c+d*e+f*g*. You might have noted that there is no parenthesis. In such expressions when there is addition and multiplication together, we have to decide which operation should be performed first. At that time, we have talked about the operator precedence. While converting infix expression into postfix expression, we have written a routine, which tells about the precedence of the operators like multiplication has higher precedence than the addition. In case of the expression *2 + 3 * 4*, we first evaluate *3 * 4* before adding *2* to the result. We are used to solve such expressions and know how to evaluate such expressions. But in the computers, we have to set the precedence in our functions.

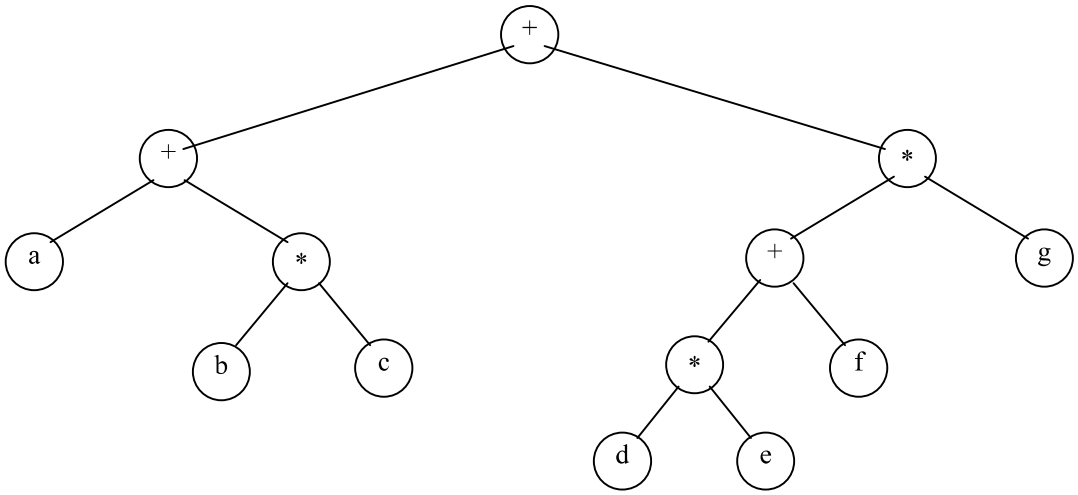
We have an expression tree and perform inorder traversal on it. We get the infix form of the expression with the inorder traversal but without parenthesis. To have the parenthesis also in the expressions, we will have to do some extra work.

Here is the code of the inorder routine which puts parenthesis in the expression.

```
/* inorder traversal routine using the parenthesis */
void inorder(TreeNode<int>* treeNode)
{
  if( treeNode != NULL )
  {
   if(treeNode->getLeft() != NULL && treeNode->getRight() != NULL) //if not leaf
     cout<<"(";
   inorder(treeNode->getLeft());
   cout << *(treeNode->getInfo())<<" ";
   inorder(treeNode->getRight());
   if(treeNode->getLeft() != NULL && treeNode->getRight() != NULL) //if not leaf
     cout<<")";
   }
}
```

This is the same inorder routine used by us earlier. It takes the root of the tree to be traversed. First of all, we check that the root node is not null. In the previous routine after the check, we have a recursive call to inorder passing it the left node, print the *info* and then call the inorder for the right node. Here we have included parenthesis using the *cout* statements. We print out the opening parenthesis '(' before the recursive call to inorder. After this, we close the parenthesis. Then we print the *info* of the node and again have opening parenthesis and recursive call to inorder with the right node before having closing parenthesis in the end. You must have understood that we are using the parenthesis in a special order. We want to put the opening parenthesis before the start of the expression or sub expression of the left node. Then we close the parenthesis. So inside the parenthesis, there is a sub expression.

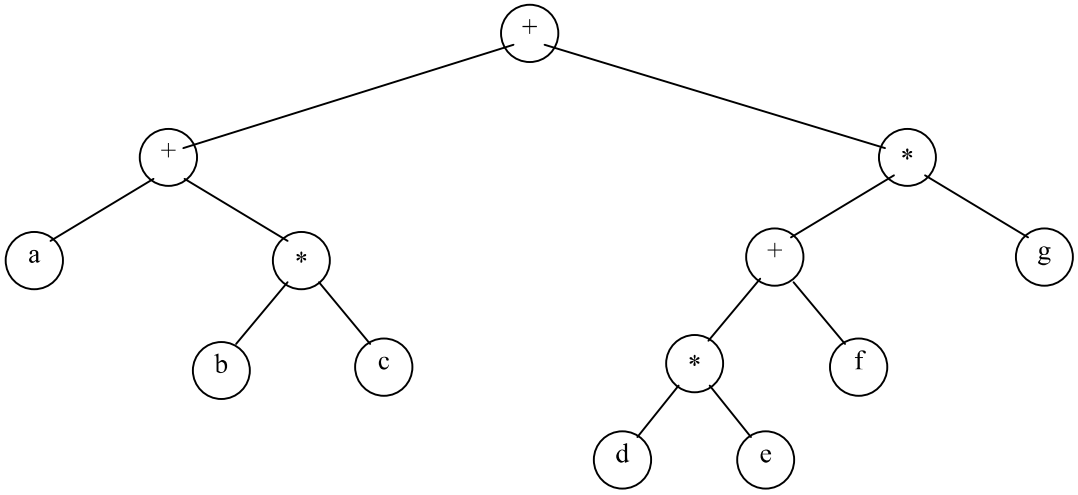
On executing this inorder routine, we have the expression as *(a + (b * c)) + (((d * e) + f) * g)*.



Inorder: *(a + (b * c)) + (((d * e) + f) * g)*

We put an opening parenthesis and start from the root and reach at the node '*a*'. After reaching at plus (+), we have a recursive call for the subtree *. Before this recursive call, there is an opening parenthesis. After the call, we have a closing parenthesis. Therefore the expression becomes as *(a + (b * c))*. Similarly we recursively call the right node of the tree. Whenever, we have a recursive call, there is an opening parenthesis. When the call ends, we have a closing parenthesis. As a result, we have an expression with parenthesis, which saves a programmer from any problem of precedence now.

Here we have used the inorder traversal. If we traverse this tree using the postorder mode, then what expression we will have? As a result of postorder traversal, there will be postorder expression.



Postorder traversal: a b c * + d e * f + g * +

This is the same tree as seen by us earlier. Here we are performing postorder traversal. In the postorder, we print left, right and then the parent. At first, we will print *a*. Instead of printing +, we will go for *b* and print it. This way, we will get the postorder traversal of this tree and the postfix expression of the left side is *a b c* * + while on the right side, the postfix expression is *d e* * *f* + *g* * +. The complete postfix expression is *a b c* * + *d e* **f* + *g* * +. The expression undergoes an alteration with the change in the traversal order. If we have some expression tree, there may be the infix, prefix and postfix expression just by traversing the same tree. Also note that in the postfix form, we do not need parenthesis.

Let's see how we can build this tree. We have some mathematical expressions while having binary operators. We want to develop an algorithm to convert postfix expression into an expression tree. This means that the expression should be in the postfix form. In the start of this course, we have seen how to covert an infix expression into postfix expression. Suppose someone is using a spreadsheet program and typed a mathematical expression in the infix form. We will first convert the infix expression into the postfix expression before building expression tree with this postfix expression.

We already have an expression to convert an infix expression into postfix. So we get the postfix expression. In the next step, we will read a symbol from the postfix expression. If the symbol is an operand, put it in a tree node and push it on the stack. In the postfix expression, we have either operators or operands. We will start reading the expression from left to right. If the symbol is operand, make a tree node and push it on the stack. How can we make a tree node? Try to memorize the *treeNode* class. We pass it some data and it returns a *treeNode* object. We insert it into the tree. A programmer can also use the *insert* routine to create a tree node and put it in the tree.

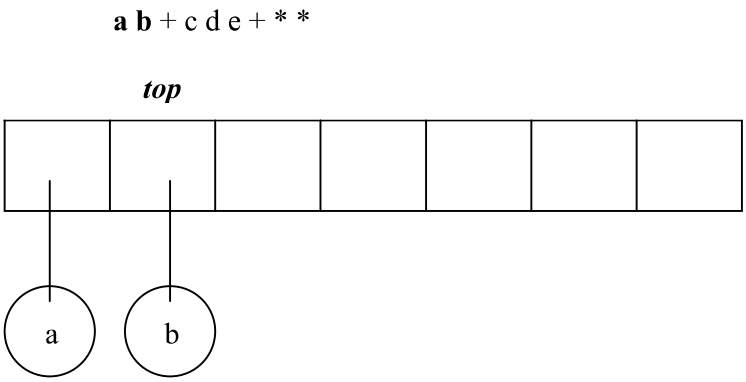
Here we will create a tree node of the operand and push it on the stack. We have been using templates in the stack examples. We have used different data types for stacks like numbers, characters etc. Now we are pushing *treeNode* on the stack. With the help of templates, any kind of data can be pushed on the stack. Here the data type of

the stack will be *treeNode*. We will push and pop elements of type *treeNode* on the stack. We will use the same stack routines.

If symbol is an operator, pop two trees from the stack, form a new tree with operator as the root and *T1* and *T2* as left and right subtrees and push this tree on the stack. We are pushing operands on the stacks. After getting an operator, we will pop two operands from the stack. As our operators are binary, so it will be advisable to pop two operands. Now we will link these two nodes with a parent node. Thus, we have the binary operator in the parent node.

Let's see an example to understand it. We have a postfix expression as *a b + c d e + * **. If you are asked to evaluate it, it can be done with the help of old routine. Here we want to build an expression tree with this expression. Suppose that we have an empty stack. We are not concerned with the internal implementation of stack. It may be an array or link list.

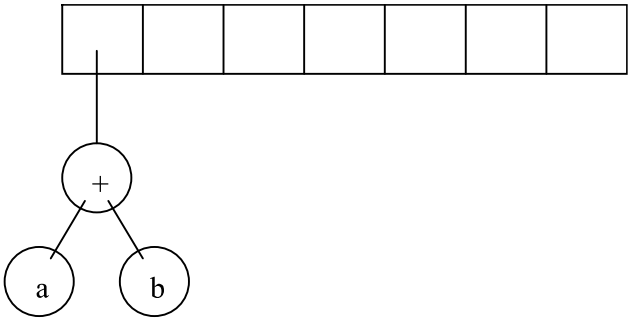
First of all, we have the symbol *a* which *is* an operand. We made a tree node and push it on the stack. The next symbol is *b.* We made a tree node and pushed it on the stack. In the below diagram, stack is shown.



If symbol is an operand, put it in a one node tree and push it on a stack.

Our stack is growing from left to right. The top is moving towards right. Now we have two nodes in the stack. Go back and read the expression, the next symbol is + which is an operator. When we have an operator, then according to the algorithm, two operands are popped. Therefore we pop two operands from the stack i.e. *a* and *b*. We made a tree node of +. Please note that we are making tree nodes of operands as well as operators. We made the + node parent of the nodes *a* and *b*. The left link of the node + is pointing to *a* while right link is pointing to *b*. We push the + node in the stack.

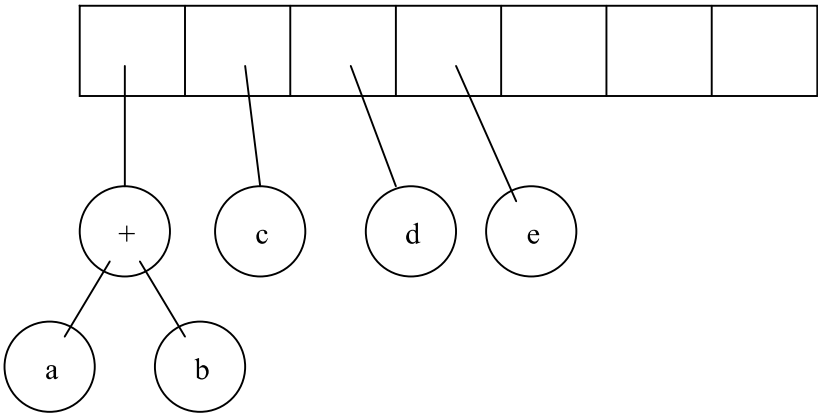
a b + c d e + * *



If symbol is an operator, pop two trees from the stack, form a new tree with operator as the root and T_1 and T_2 as left and right subtrees and push this tree on the stack.

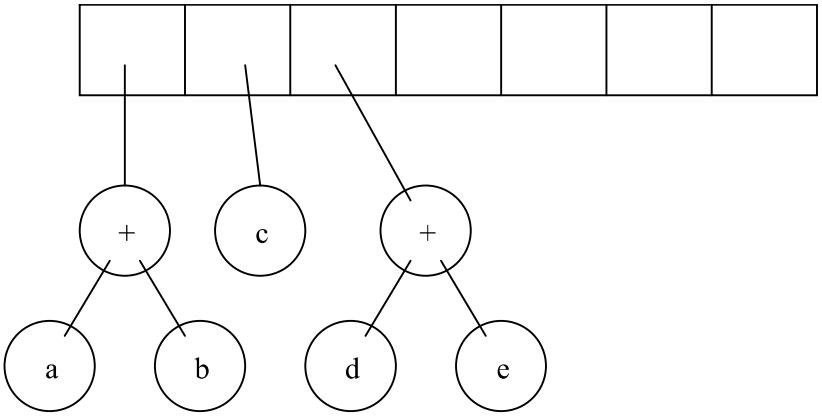
Actually, we push this subtree in the stack. Next three symbols are *c, d,* and *e*. We made three nodes of these and push these on the stack.

a b + c d e + * *



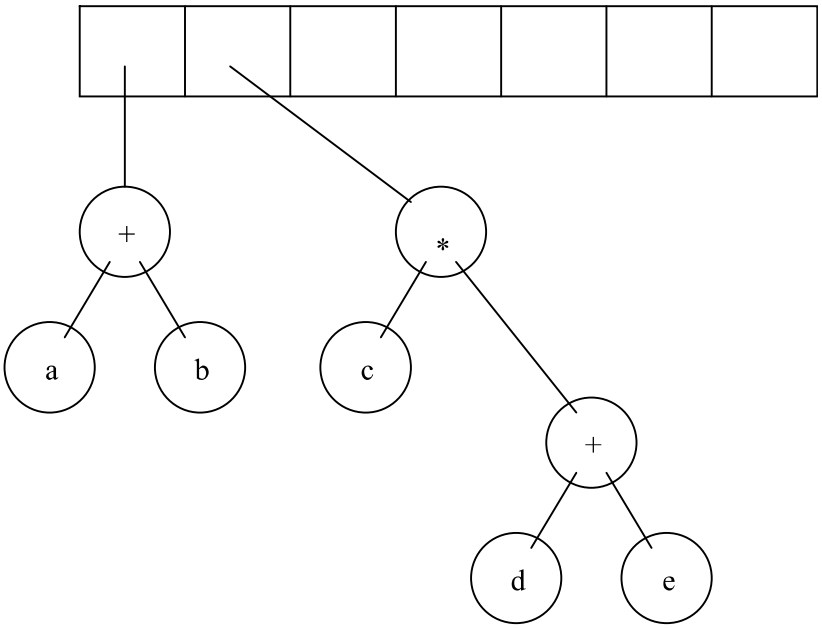
Next we have an operator symbol as +. We popped two elements i.e. *d* and *e* and linked the + node with *d* and *e* before pushing it on the stack. Now we have three nodes in the stack, first + node under which there are *a* and *b*. The second node is *c* while the third node is again + node with *d* and *e* as left and right nodes.

a b + c d e + * *

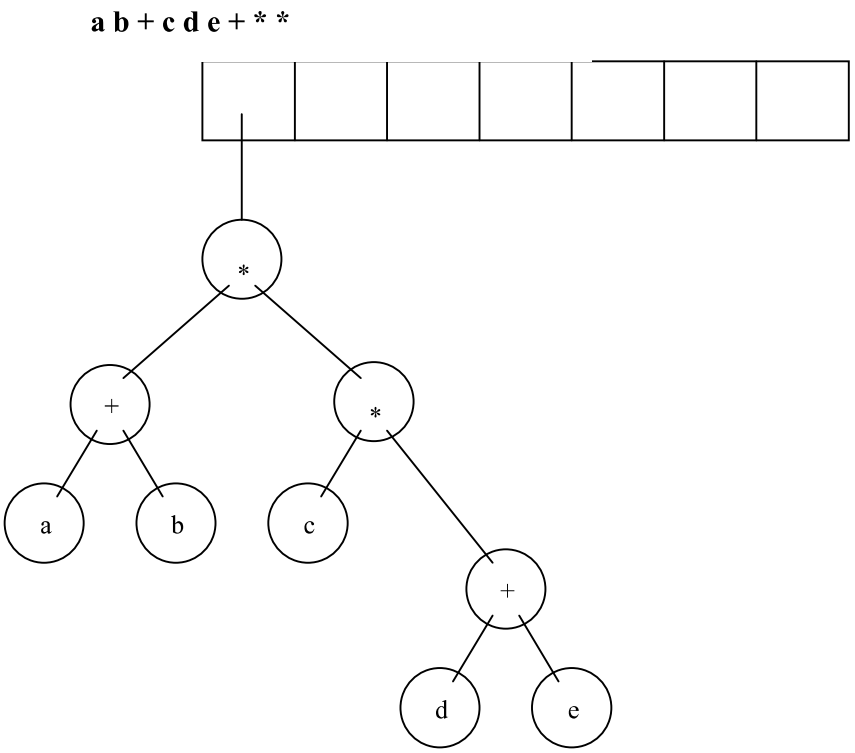


The next symbol is * which is multiplication operator. We popped two nodes i.e. a subtree of + node having *d* and *e* as child nodes and the *c* node. We made a node of * and linked it with the two popped nodes. The node *c* is on the left side of the * node and the node + with subtree is on the right side.

a b + c d e + * *



The last symbol is the * which is an operator. The final shape of the stack is as under:



In the above figure, there is a complete expression tree. Now try to traverse this tree in the inorder. We will get the infix form which is *a + b * c * d + e*. We don't have parenthesis here but can put these as discussed earlier.

This is the way to build an expression tree. We have used the algorithm to convert the infix form into postfix form. We have also used stack data structure. With the help of templates, we can insert any type of data in the stack. We have used the expression tree algorithm and very easily built the expression tree.

In the computer science, trees like structures are used very often especially in compilers and processing of different languages.

Huffman Encoding

There are other uses of binary trees. One of these is in the compression of data. This is known as Huffman Encoding. Data compression plays a significant role in computer networks. To transmit data to its destination faster, it is necessary to either increase the data rate of the transmission media or simply send less data. The data compression is used in computer networks. To make the computer networks faster, we have two options i.e. one is to somehow increase the data rate of transmission or somehow send the less data. But it does not mean that less information should be sent or transmitted. Information must be complete at any cost.

Suppose you want to send some data to some other computer. We usually compress the file (using winzip) before sending. The receiver of the file decompresses the data before making its use. The other way is to increase the bandwidth. We may want to use the fiber cables or replace the slow modem with a faster one to increase the