**Expression trees**

We will create a binary tree, not a Binary Search Tree.

The construction starts with reading the postfix expression one symbol at a time. To do this we need a tree structure and a stack that holds trees for each of its items.

struct tree{

char item;

tree \*left;

tree \*right;

};

//each element of the stack is a tree.

struct stacktree{

struct tree \*item;

stacktree \*previous;

};

If the symbol is an operand, we create a one-node tree and push it onto a stack.

If the symbol is an operator, we pop two trees T1 and T2 from the stack (popping up T1 first), and form a new tree with the operator as the root, T2 as the left child and T1 as the right child. The new tree is then pushed onto the stack. As an example, let the input expression be

a b +

The first two symbols are operands. So we create one node trees and push them onto a stack.

|  |  |
| --- | --- |
| Stack  tree |  |

The next symbol is an operator +, so the two one node trees are popped, and a new tree is formed. The first tree to be popped is the one containing b, so this becomes the right child of the operator, the second tree to be popped contains the operand a , and this becomes the left child of the operator.



Let us take a larger expression:

a b + c d e + \* \*

Let us say we have processed the first three tokens, and have already reached the ‘+’ operator. a b +

Next we read c, d, e and construct one node trees and push them onto the stack.



Next is an operator ‘+’, so last two trees are popped out and a new tree formed with this as root.



Next symbol is another operator ‘\*’, so again two trees are popped from the stack and merged to form a new tree with this operator at the root:



Finally the last operator ‘\*’ causes the last two entries from the stack to be pulled out , i.e. the tree containing ‘+’ at the root and the tree containing ‘\*’ at the root. These are merged with the second ‘\*’ operator as the root, and the expression tree is complete, and now it is the only entry in the stack.



Traverse the tree in postorder: **a b + c d e + \* \***

When an infix expression is printed, an opening and closing parenthesis must be added at the beginning and ending of each expression. As every subtree represents a subexpression, an opening parenthesis is printed at its start and the closing parenthesis is printed after processing all of its children.

**((a+b)\* (c\*(d+e))**

**Evaluating the tree**

int postOrder(struct tree \*item){

int result;

if (item==NULL) return 0;

int left=postOrder(item->left);

int right=postOrder(item->right);

if (isOperator(item->value))

return (evaluate(left, right, item->value)); //evaluate expression

else

return item->value; //convert to a number

}



**Non-tail recursive algorithm**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| PO(\*)  left=PO(+) 3  right=PO(\*)27  return (3\*27)=81 | PO(+)  Left=PO(1) 1  Right=PO(2) 2  Return (1+2)=3 | PO(1)  Left=PO(NULL)  Right=PO(NULL)  Return(1) | | PO(NULL)  Return 0 |
| PO(NULL)  Return 0 |
| PO(2)  Left=PO(NULL)  Right=PO(NULL)  Return 2 | | PO(NULL)  Return 0 |
| PO(NULL)  Return 0 |
| PO(\*)  Left=PO(3)3  Right=PO(+) 9  Return (3\*9)=27 | PO(3)  Left=PO(NULL)  Right=PO(NULL)  Return(3) | | PO(NULL)  Return 0 |
| PO(NULL)  Return 0 |
| PO(+)  Left=PO(4)4  Right=PO(5)5  Return (4+5)=9 | PO(4) Left=PO(NULL) Right=PO(NULL)  Return(4) | PO(NULL)  Return 0 |
| PO(NULL)  Return 0 |
| PO(5)  Left=PO(NULL) Right=PO(NULL)  Return 5 | PO(NULL)  Return 0 |
| PO(NULL)  Return 0 |