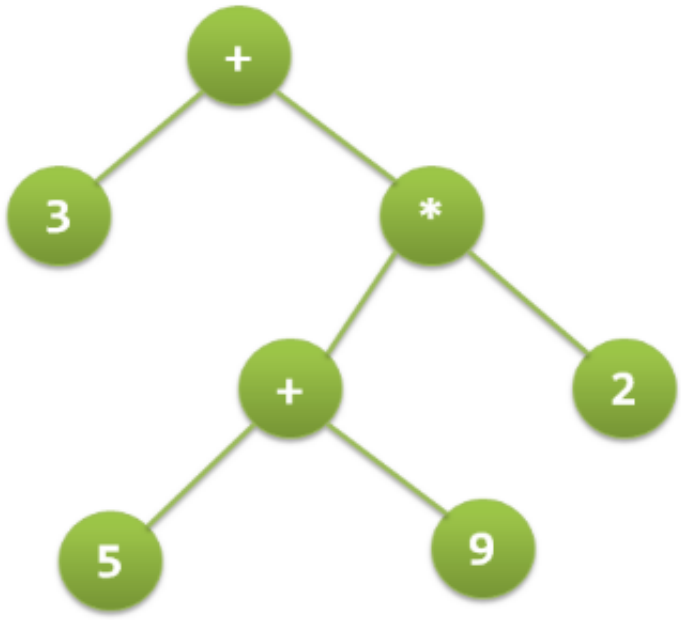
Expression Tree

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be:



Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

Evaluating the expression represented by expression tree:

```
Let t be the expression tree
If  t is not null then
      If t.value is operand then
                Return  t.value
      A = solve(t.left)
      B = solve(t.right)

      // calculate applies operator 't.value'
      // on A and B, and returns value
```

```
        Return calculate(A, B, t.value)
```

Construction of Expression Tree:

Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.

1) If character is operand push that into stack

2) If character is operator pop two values from stack make them its child and push current node again.

At the end only element of stack will be root of expression tree.

Below is the implementation :

C/C++

```
// C++ program for expression tree
#include<bits/stdc++.h>
using namespace std;

// An expression tree node
struct et
{
    char value;
    et* left, *right;
};

// A utility function to check if 'c'
// is an operator
bool isOperator(char c)
{
    if (c == '+' || c == '-' ||
            c == '*' || c == '/' ||
            c == '^')
        return true;
    return false;
}

// Utility function to do inorder traversal
void inorder(et *t)
{
    if(t)
    {
        inorder(t->left);
        printf("%c ", t->value);
        inorder(t->right);
    }
}

// A utility function to create a new node
et* newNode(int v)
{
    et *temp = new et;
    temp->left = temp->right = NULL;
    temp->value = v;
    return temp;
};

// Returns root of constructed tree for given
// postfix expression
et* constructTree(char postfix[])
{
```

```
    stack<et *> st;
    et *t, *t1, *t2;

    // Traverse through every character of
    // input expression
    for (int i=0; i<strlen(postfix); i++)
    {
        // If operand, simply push into stack
        if (!isOperator(postfix[i]))
        {
            t = newNode(postfix[i]);
            st.push(t);
        }
        else // operator
        {
            t = newNode(postfix[i]);

            // Pop two top nodes
            t1 = st.top(); // Store top
            st.pop();      // Remove top
            t2 = st.top();
            st.pop();

            //  make them children
            t->right = t1;
            t->left = t2;

            // Add this subexpression to stack
            st.push(t);
        }
    }

    //  only element will be root of expression
    // tree
    t = st.top();
    st.pop();

    return t;
}

// Driver program to test above
int main()
{
    char postfix[] = "ab+ef*g*-";
    et* r = constructTree(postfix);
    printf("infix expression is \n");
    inorder(r);
    return 0;
}
```

Run on IDE

Java

```
// Java program to construct an expression tree

import java.util.Stack;

// Java program for expression tree
class Node {

    char value;
```

```
    Node left, right;

    Node(char item) {
        value = item;
        left = right = null;
    }
}

class ExpressionTree {

    // A utility function to check if 'c'
    // is an operator

    boolean isOperator(char c) {
        if (c == '+' || c == '-'
                || c == '*' || c == '/'
                || c == '^') {
            return true;
        }
        return false;
    }

    // Utility function to do inorder traversal
    void inorder(Node t) {
        if (t != null) {
            inorder(t.left);
            System.out.print(t.value + " ");
            inorder(t.right);
        }
    }

    // Returns root of constructed tree for given
    // postfix expression
    Node constructTree(char postfix[]) {
        Stack<Node> st = new Stack();
        Node t, t1, t2;

        // Traverse through every character of
        // input expression
        for (int i = 0; i < postfix.length; i++) {

            // If operand, simply push into stack
            if (!isOperator(postfix[i])) {
                t = new Node(postfix[i]);
                st.push(t);
            } else // operator
            {
                t = new Node(postfix[i]);

                // Pop two top nodes
                // Store top
                t1 = st.pop();      // Remove top
                t2 = st.pop();

                //  make them children
                t.right = t1;
                t.left = t2;

                // System.out.println(t1 + "" + t2);
                // Add this subexpression to stack
                st.push(t);
            }
        }

        //  only element will be root of expression
```

```
        // tree
        t = st.peek();
        st.pop();

        return t;
    }

    public static void main(String args[]) {

        ExpressionTree et = new ExpressionTree();
        String postfix = "ab+ef*g*-";
        char[] charArray = postfix.toCharArray();
        Node root = et.constructTree(charArray);
        System.out.println("infix expression is");
        et.inorder(root);

    }
}

// This code has been contributed by Mayank Jaiswal
```

Run on IDE

Python

```
# Python program for expression tree

# An expression tree node
class Et:

    # Constructor to create a node
    def __init__(self , value):
        self.value = value
        self.left = None
        self.right = None

# A utility function to check if 'c'
# is an operator
def isOperator(c):
    if (c == '+' or c == '-' or c == '*'
        or c == '/' or c == '^'):
        return True
    else:
        return False

# A utility function to do inorder traversal
def inorder(t):
    if t is not None:
        inorder(t.left)
        print t.value,
        inorder(t.right)

# Returns root of constructed tree for
# given postfix expression
def constructTree(postfix):
    stack = []

    # Traverse through every character of input expression
    for char in postfix :

        # if operand, simply push into stack
        if not isOperator(char):
```

```
            t = Et(char)
            stack.append(t)

        # Operator
        else:

            # Pop two top nodes
            t = Et(char)
            t1 = stack.pop()
            t2 = stack.pop()

            # make them children
            t.right = t1
            t.left = t2

            # Add this subexpression to stack
            stack.append(t)

    # Only element  will be the root of expression tree
    t = stack.pop()

    return t

# Driver program to test above
postfix = "ab+ef*g*-"
r = constructTree(postfix)
print "Infix expression is"
inorder(r)
```

Run on IDE

Output:

```
infix expression is
a + b - e * f * g
```

This article is contributed by **Utkarsh Trivedi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

9 Comments Category: Trees

Related Posts:

- Check sum of Covered and Uncovered nodes of Binary Tree
- Lowest Common Ancestor in a Binary Tree | Set 2 (Using Parent Pointer)
- Construct a Binary Search Tree from given postorder
- BFS vs DFS for Binary Tree
- Maximum difference between node and its ancestor in Binary Tree
- Inorder Non-threaded Binary Tree Traversal without Recursion or Stack
- Check if leaf traversal of two Binary Trees is same?
- Closest leaf to a given node in Binary Tree

(Login to Rate and Mark)

2.8 Average Difficulty : **2.8/5.0**
Based on **7** vote(s)

Add to TODO List
Mark as DONE

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

9 Comments GeeksforGeeks 1 Login

Recommend 4 Share Sort by Newest

Join the discussion…

Rohit Sharma • 19 days ago

Expression tree using prefix expression

```
struct et* constructTree_from_prefix(char* prefix, int* index) {

if(prefix[*index] == '\0')

return NULL;

struct et* node=NULL;

if(isOperator(prefix[*index])) {

node = newNode(prefix[*index]);
```

```
node = newNode(prefix[*index]);

(*index)++;

if(prefix[*index] != '\0') {

node->left = constructTree_from_prefix(prefix, index);

node->right = constructTree_from_prefix(prefix, index);
```

see more

• Reply • Share ›

ddd • 5 months ago

http://csis.pace.edu/~wolf/CS1...

• Reply • Share ›

Prajwal Muralidhara • 5 months ago

How will the implementation change for a unary operator ? (like increment operator ?)

• Reply • Share ›

Rushikesh Deshpande ➚ Prajwal Muralidhara • 4 months ago

http://stackoverflow.com/quest...

• Reply • Share ›

Kumar Pawan • 7 months ago

In c It can be implemented as http://crazycoders.quora.com/P....

• Reply • Share ›

Rajnish Kr Jha • 7 months ago

return temp is missing in newNode function.
And in code you are popping element from stack while stack is empty ,this may lead to an undefined outcome and may throw std::logic_error .
I just modified the code a little bit and it is giving desired output .

http://ideone.com/C3UyGD

3 • Reply • Share ›

Mayank ➚ Rajnish Kr Jha • 7 months ago

Rajnish, your point about empty stack is valid only if the input postfix expression is invalid. In a valid postfix expression we will always have two operands before an operator.

1 • Reply • Share ›

Rajnish Jha • 7 months ago

In newNode function , return temp is missing !

1 • Reply • Share ›

GeeksforGeeks Mod Rajnish Jha • 7 months ago

Thanks for pointing this out. We have added return statement.

• Reply • Share ›