ASSIGNMENT NO.:- A-3

AIM: To study conversion of given inorder expression into binary tree.

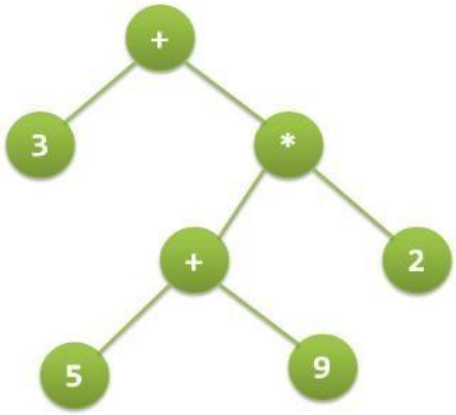
INDEX TERMS: class, objects, keyword, infix, postfix, Expression conversion into binary tree.

PROBLEM DEFINITION:

For given expression eg. a-b*c-d/e+f construct inorder sequence and traverse it using postorder traversal(non recursive).

THEORY:

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 ,same with preorder traversal it gives prefix expression.

In this assignment we will be accepting infix expression from user then convert it into postfix and from postfix construct the Binary tree using stack. Finally traverse the tree using inorder, preorder and postorder traversal and print the results.

Algorithm to push element into stack

```
push(charval)
{
    1.      if(top==max-1)
    2.      {
    3.      cout<<"\n Stack Overflow";
    4.      }
    5.      else
    6.      {
```

```
    7.      data[++top]=val;
    8.      cout<<"\n Data added into stack"<<val;
    9.      }
}
```

Algorithm to pop element into stack

```
pop()
{
    1. if(top<0)
    2. {
    3. cout<<"\n Stack Is Underflow";
    4. }
    5. else
    6. {
    7. cout<<"\n"<<data[top--]<<"Stack Is Poped";

    8. }
}
```

Algorithm to convert infix expression to the postfix notation.

1) Scan the given expression from left to right.
2) First operator seen is simply pushed onto stack.
3) If we see an operand, append it to the postfix expression.
4) If we see an operator (x), pop off all the operators which are of lower precedence than 'x' and append them to the postfix expression. Then, push the operator 'x' onto stack.
5) If we see an opening parenthesis, simply push it onto stack.
6) If we see a closing parenthesis, pop off all elements from stack till opening parenthesis and append them topostfix expression except the opening & closing parenthesis.
7) Finally, pop off all the elements (operators) from stack till it's empty and append them to postfix expression.

Algorithm to evaluate a postfix expression.

1) Scan the expression from left to right.
2) If an operand is seen, push it onto stack.
3) If an operator is seen, pop of top two elements (operands) [x & y] from stack and perform z = x operand y . Push z onto stack.
4) Repeat steps (2) & (3) till scanning is over.

Analysis

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- **Push Operation** : O(1)
- **Pop Operation** : O(1)
- **Top Operation** : O(1)
- **Search Operation** : O(n)

Construction of Expression Tree from Postfix :

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.

IMPLEMENTATION

```
// An expression tree node class
node
{
public:
    char            value;
    node* 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(node *t)
```

```
{
        if(t)
        {
             inorder(t->left);
     printf("%c ", t->value);
     inorder(t->right);
        }
}
// A utility function to create a new node node*
newNode(int v)
{
     node *temp = new node;
     temp->left = temp->right = NULL;
     temp->value = v;   return temp;
};

// Returns root of constructed tree for given postfix expression
node* constructTree(char postfix[])
{
     stack<node *> st;
     node *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*-";
     node* r = constructTree(postfix);
     cout<<"infix expression is \n";
     inorder(r);    return 0;
}
```

FAQ:

1. What is time complexity of program in all cases?
2. How expression trees are gets represented in data structure?