```c
/* 8. Develop and execute a program in C using suitable data structures to create
       a binary tree for a expression. The tree traversals in some proper method
       should result in conversion of original expression into prefix, infix and
       postfix forms. Display the original expression along with the three
       different forms also. */

// Develop and execute a program in C using suitable data structures to create a
//      binary tree for a expression

// Convert a infix expression, consisting of numbers and arithemric operators,
//      to a binary expression tree; with binary operators : works with two operands

// The shunting-yard algorithm is a method for parsing mathematical expressions
//    specified in infix notation
// It can produce either a postfix notation string, also known as Reverse Polish
//    notation (RPN), or abstract syntax tree (AST); AST is Binary expression tree

// The algorithm was invented by Edsger Dijkstra and named the "shunting yard"
//    algorithm because its operation resembles that of a railroad shunting yard

// Like the evaluation of RPN, the shunting yard algorithm is stack-based
// Infix expressions are the form of mathematical notation like
// 3 + 4 or 3 + 4 * (2 - 1)

// To convert, the program reads each symbol in order
//      Then apply shunting-yard algorithm based on the symbol

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

struct node // Binary tree node
 {
   struct node *left;   // left child
   char data;           // operator or operand
   struct node *right;  // right child
 };

char expression[256]; // Save arithmetic expression

char operatorStack[128]; // Stack: holds arithmetic operators
int  operatorStackTop = -1;

struct node* outputStack[128];//Nodes/tree of Operands or sub expressions address
int    outputStackTop = -1;

/* Suppose a function buildExpression() is :
       op = pop operator from the operator stack
       subExp2 = pop from output stack // pop twice from output stack
       subExp1 = pop from output stack
       build tree with op as root and subExp1 and subExp2 as left and right
           children respectively
       push result on result stack */

/* Shunting-Yard Algorithm

   while there are tokens to be read: // token : a character from input string
       read a token // character can be a operator or a operand

       if the token is a number/alphabet, then push it to the output stack

       if the token is a left bracket "(", then:
          push it onto the operator stack

       if the token is a right bracket ")", then:
          while the operator at the top of the operator stack is not a "(" :
              buildExpression()

          then pop the left bracket from the stack
```

```
        if the token is an operator, then:
           while there is an operator at the top of the operator stack
                 with >= precedence :
                    buildExpression()

           then push the read operator onto the operator stack

        if the stack runs out without finding a left bracket, then there are
           mismatched parentheses , input is invalid arithmetic expression

    if there are no more tokens to read:
        while there are still operator tokens on the stack:
            if the operator token on the top of the stack is a bracket, then
                 there are mismatched parentheses , input is invalid arithmetic
                 expression

            else buildExpression()

    then pop from output stack, return this result
    which will be the address of root element of expression tree */

 // Assumption : Input is valid arithmetic expression
 //              Operators can be + - * or /
 //              Operands can be single character or digit

 struct node* createNode(char value)
  { // allocate space for new node, ewnode holds address if malloc successful
    struct node *newNode = (struct node*)malloc(sizeof(struct node));

    newNode -> data = value; // value is either operand or operator
    newNode -> left = NULL;
    newNode -> right = NULL; // Initialise newnode's left and right node with NULL

    return newNode;
  }

 void pushIntoOutputStack ( struct node *newNode )//Push address of result tree
  {//with operator as root, subexpressions/operands as children on output stack
    outputStack [ ++outputStackTop ] = newNode ;
  }

 struct node* popFromOutputStack() // pop address of result tree from output stack
  {
    return outputStack [ outputStackTop-- ];
  }

 void pushIntoOperatorStack( char operation ) // push operator on operator stack
  {
    operatorStack[++operatorStackTop] = operation ;
  }

 char popFromOperatorStack() // pop operator from operator stack
  {
    return operatorStack [ operatorStackTop-- ];
  }

 void buildExpression() // build expression tree with operator from operator stack
  { // and results from output stack
    char operation = popFromOperatorStack();//op= pop from the operator stack
    struct node *subExp2 = popFromOutputStack(); // subExp2 = pop from output stack
    struct node *subExp1 = popFromOutputStack(); // subExp1 = pop from output stack

    struct node *newNode = createNode( operation ); // build tree with op as root
    newNode -> left  = subExp1 ; //  and subExp1 and subExp2 as left and right
    newNode -> right = subExp2 ; //  children respectively

    pushIntoOutputStack( newNode );  // push result on result stack
  }
```

```c
void printErrorMessage()
 {
   printf("\n Invalid Expression or Expression should have single character");
   printf(" or digit operand or + - * / as operators\n");
 }

int precedence( char operation ) // return precedence of operators
 { // why give '(' open parenthesis lesser precedence in program implementation ?
   switch (operation) // but in BODMAS, Bracket has highest precedence
    {
      case '(': return 0;
      case '+':
      case '-': return 1;  // lower
      case '/':
      case '*': return 2;  // higher precedence
    }
 }

struct node* infixToBinaryTree( char *expression ) // return address of binary tree
 { // representation of expression
   int i=0;

   while ( expression[i] != '\0' ) // while there are tokens to be read
    {
      if ( expression[i] == ' ' )  ; // over look white space
      else if ( isdigit( expression[i] ) || isalpha( expression[i] ) )
         // if the token is a number/alphabet, then push it to the output stack
         pushIntoOutputStack ( createNode( expression[i] ) ) ;
      else if ( expression[i] == '(' ) // if the token is a left bracket "(", then:
         pushIntoOperatorStack( expression[i] ); // push it onto the operator stack
      else if ( expression[i] == ')' )//if the token is a right bracket ")", then:
       {
         int j = operatorStackTop;
         while ( operatorStack[j] != '(' && j >= 0 )//while the operator at the top
          { //  of the operator stack is not a left bracket:
            buildExpression();
            j--;
          } // if( j < 0 ) { printErrorMessage(); exit(1); }
         char temp = popFromOperatorStack(); // pop the left bracket from the stack
       }
      else if ( expression[i] == '+' || expression[i] == '-' ||
                expression[i] == '*' || expression[i] == '/'     )
       { // if the token is an operator, then: check if operatorStackTop is valid
         while ( operatorStackTop != -1 && // and while there is operator at top
                 precedence( operatorStack[operatorStackTop] ) >= // of operator
                   precedence( expression[i] ) ) // stack with >= precedence : then
            buildExpression();

         pushIntoOperatorStack(expression[i]);//push read operator on operatorStack
       }
      else
       {
         printErrorMessage(); exit(1);
       }
      i++;
    }
   // if there are no more tokens to read:
   while ( operatorStackTop != -1 ) // while there are still operator tokens
      buildExpression();            //  on the operator stack: build expression

   if( outputStackTop == 0 )//if expression is valid, then only expression tree's
    { // root address will be on stack
      return ( popFromOutputStack() );//pop from output stack, return resultAddress
    }
   else
    {
      printErrorMessage(); exit(1);
    }
```

```
 }

void preOrder(struct node *root) // The tree traversals in some proper method
should
 {                                // result in conversion of original expression
   if( root != NULL )             // into prefix, infix and postfix forms
    {                             // Preorder gives prefix
      printf("%c ",root->data);     // Print node
      preOrder(root->left);         // Process left sub tree
      preOrder(root->right);        // Process right sub tree
    }
 }

void inOrder(struct node *root)  // Inorder traversal of AST gives
 { //  infix notation of corresponding infix expression
   if( root != NULL )
    {
      inOrder(root->left);           // Process left sub tree
      printf("%c ",root->data);      // Print node
      inOrder(root->right);          // Process right sub tree
    }
 }

void postOrder(struct node *root) // Postorder gives postfix
 {
   if( root != NULL )
    {
      postOrder(root->left);          // Process left sub tree
      postOrder(root->right);         // Process right sub tree
      printf("%c ",root->data);       // Print node
    }
 }

int main()
 {
   printf("\n Expression should have single character or digit as operand ");
   printf("and + - * / operators\n Enter valid arithmetic expression : ");
   scanf("%[^\n]", expression); // Why format specifier as %[^\n] ?

   struct node *ast = infixToBinaryTree( expression );

   printf("\n Original expression = %s\n", expression);
   // Display original expression
   printf("\n Prefix = "); // along with the three different forms also
   preOrder(ast);

   printf("\n Infix = ");
   inOrder(ast);

   printf("\n Postfix = ");
   postOrder(ast);

   return(0);
 }
/* Output :
 Expression should have single character or digit as operand and + - * / operators
 Enter valid arithmetic expression : (a + b * c) + ((d * e + f ) * g)
 Original expression = (a + b * c) + ((d * e + f ) * g)
 Prefix = + + a * b c * + * d e f g
 Infix = a + b * c + d * e + f * g
 Postfix = a b c * + d e * f + g * +
*/
```