//expv2lf.h

```
typedef struct
{
        char c[100];
        int top;
        int limit;
}stack;

struct node
{
        char a;
        struct node *left,*right;
};

typedef struct node node;

typedef struct
{
        node *c[100];
        int top;
        int limit;
}stackAd;

typedef struct
{
        stack s;
        stackAd t;
        node *p;
        char infix[100],postfix[100];
        int value;
}expADT;

void initialise(stack *t);//function to initialise stack members
int isempty(stack *t);//function to check if stack is empty
int isfull(stack *t);//function to check if stack is full
void Size(stack t);//function to return the size of the stack
void disp(stack t);//function to display stack
void push(stack *t,char x);//function to push x into stack t
char pop(stack *t);//function to pop an element from the stack
char * infixtoPostfix(char *str, stack *s);//function to convert infix expression to postfix expression
int evaluateExp(char *postfix, stack *s);//function to evaluate a postfix expression

void initialisenAd(stackAd *t);//stack function for address stack
int isemptyAd(stackAd *t);
int isfullAd(stackAd *t);
void pushAd(stackAd *t,node *x);
node* popAd(stackAd *t);
void createtree(expADT *d);//creating a expression tree
void displaytree(node *p);//displaying the expression tree using inorder notation
int evaluatetree(node *d);//evaluation of expression using expression tree
```

//expv2Impl.h

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
#include "extADTv2if.h"

void initialisen(stack *t)              //function to initialise stack members with constant values
{
    (*t).top=-1;
    t->limit=100;
}
int isempty(stack *t)                    //function to check if stack is empty
{
    if((*t).top==-1)
        return 1;
    else
        return 0;
}

int isfull(stack *t)                     //function to check if stack is full
{
    if((*t).top==(*t).limit-1)
        return 1;
    else
        return 0;
}

void Size(stack t)                           //function to return the size of the stack
{
    printf("\nThe size of the given stack is %d",t.top+1);
}

void disp(stack t)                           //function to display stack
{
    int i;
    for(i=0;i<=t.top;i++)
        {
        printf("\n%c",t.c[i]);
        if(i==t.top)
```

```
                        printf(" <--");
                }
}

void push(stack *t,char x)                     //function to push x into stack t
{
        if(isfull(t))
                printf("\n The stack is full");
        else
                {
                (*t).top++;
                t->c[(*t).top]=x;

                }

}

char pop(stack *t)                     //function to pop elements
{
        if(isempty(t))
                return 0;
        else
                {
                char x=t->c[(*t).top];
                (*t).top--;
                return x;
                }
}

int prec(char c)                       //function to check precedence of operators
{
    if(c == '^')
        return 3;
    else if(c == '*' || c == '/')
        return 2;
    else if(c == '+' || c == '-')
        return 1;
    else
        return -1;
}

char *infixtoPostfix(char *str, stack *s) //function to convert infix expression to postfix expression
{
            push(s,1);
            static char st[100];
            int j=0;
            for(int i=0;i<strlen(str);i++)
            {
                        char x=*(str+i);

                        if(x!='+'&&x!='-'&&x!='*'&&x!='/'&&x!=')'&&x!='('&&x!='^')         //if its a operand
                        {
                                    st[j++]=x;

                        }
                         else if(x == '(')
                                    push(s,'(');
                        else if(x == ')')
        {
            while(s->c[(*s).top] != 1 && s->c[(*s).top] != '(')
            {
                char c = pop(s);
                st[j++]=c;


            }
            if(s->c[(*s).top] == '(')
            {
                char c = pop(s);


            }
        }
                        else{
            while(s->c[(*s).top] != 1 && prec(x) <= prec(s->c[(*s).top]))
            {
                char c = pop(s);
                st[j++]=c;


            }
           push(s,x);
        }



             }
             while(s->c[(*s).top] != 1)
    {
                                                char c = pop(s);
                st[j++]=c;
    }
             st[j++]=0;


return st;
}
```

```
int evaluateExp(char *postfix, stack *s)   //function to evaluate a postfix expression

{



    int i;


    for (i = 0; postfix[i]; ++i)

    {


        if (isdigit(postfix[i]))                         //push the operands into the stack

            push(s, postfix[i] - '0');


        else

        {

            int val1 = pop(s);

            int val2 = pop(s);

            switch (postfix[i])                          //if scanned character is a operator,pop 2 values,evaluate
and push it back

            {

            case '+': push(s, val2 + val1); break;

            case '-': push(s, val2 - val1); break;

            case '*': push(s, val2 * val1); break;

            case '/': push(s, val2/val1); break;

            }

        }

    }

    return pop(s);

}



/***************************address stack*****************************/
```

```
void initialisenAd(stackAd *t)                  //function to initialise stack members with constant values

{

            (*t).top=-1;

            t->limit=100;

}


int isemptyAd(stackAd *t)                                   //function to check if stack is empty

{

            if((*t).top==-1)

                        return 1;

            else

                        return 0;

}


int isfullAd(stackAd *t)                          //function to check if stack is full

{

            if((*t).top==(*t).limit-1)

                        return 1;

            else

                        return 0;

}


void pushAd(stackAd *t,node *x)                                   //function to push x into stack t

{

            if(isfullAd(t))

                        printf("\n The stack is full");

            else

                        {
```

```
                      (*t).top++;
                      t->c[(*t).top]=x;
                      }
}

node* popAd(stackAd *t)                                          //function to pop elements
{
           if(isemptyAd(t))
                      return NULL;
           else
                      {
                      node *x=t->c[(*t).top];
                      (*t).top--;
                      return x;
                      }
}

/*****************tree creation************************/

void createtree(expADT *d)
{
   initialisenAd(&d->t);
   int l=strlen(d->postfix);
   int i;
   for(i=0;i<l;i++)
      { //stackAd t
         if(isalnum(d->postfix[i]))
         {
            node *temp=(node *)malloc(sizeof(node));
            temp->a=d->postfix[i];  //char data
            temp->right=NULL;
```

```
                temp->left=NULL;
                pushAd(&d->t,temp);
            }
            else
            {
                node *temp=(node *)malloc(sizeof(node));
                temp->a=d->postfix[i];
                temp->right=popAd(&d->t);
                temp->left=popAd(&d->t);
                pushAd(&d->t,temp);
            }
              }
    d->p=popAd(&d->t); //last value
}

void displaytree(node *p)//inorder display-LPR
{
    if(p!=NULL){
        displaytree(p->left);
        //printing parent
        printf("%c",p->a);
        displaytree(p->right);
    }
}

/*********************tree display****************************/

int evaluatetree(node *d)
{
              if(d->left==NULL && d->right==NULL)
              {
```

```
                        return d->a-'0';
                }
                else
                {
                        char op=d->a;
                        int val1=evaluatetree(d->left);
                        int val2=evaluatetree(d->right);
                        switch (op)
                {
                        case '+': return val1+val2; break;
                        case '-': return val1-val2; break;
                        case '*': return val1*val2; break;
                        case '/': return (int)(val1/val2); break;
                }
    }

}

/***************************print hierarchy*****************/

void printtree(node *p,int depth)
{
    int i;
    for(i=0;i<depth;i++)
        printf("\t");
    printf("%c",p->a);
    printf("\n");
    if(p->left!=NULL)
        printtree(p->left,depth+1);
    if(p->right!=NULL)
        printtree(p->right,depth+1);
```

```
}

/**********************************************************/
```

//expv2Appl.c

```
#include "extADTv2impl.h"


void main()

{

   int ch;

   do

   {


      expADT t;

      initialisen(&t.s);


      printf("\nEnter the expression to be converted : ");

      scanf("%s",t.infix);

      strcpy(t.postfix,infixtoPostfix(t.infix,&t.s));


      printf("\nThe equivalent postfix expression is : %s\n",t.postfix);

      createtree(&t);

      printf("\nTree in preorder:\n");

      printtree(t.p,0);


      t.value=evaluatetree(t.p); //node *p-head node

      printf("\nValue of the given expression is : %d\n",t.value);

      printf("\nEnter 1 to continue : ");

      scanf("%d",&ch);


   }while(ch==1);
}



/*
OUTPUT:
Enter the expression to be converted : 2*3+5-2

The equivalent postfix expression is : 23*5+2-

Tree in preorder:
-
        +
                *
                        2
                        3
                5
        2

Value of the given expression is : 9

Enter 1 to continue : 0
*/
```