

STACK EVALUATION

POSTFIX EVALUATION:

Aim: To evaluate a postfix expression using stack.

Theory: The Postfix notation is used to represent algebraic expressions. The expressions written in postfix form are evaluated faster compared to infix notation as parenthesis are not required in postfix. We have discussed infix to postfix conversion. In this post, evaluation of postfix expressions is discussed. Evaluation of a postfix expression using a stack is explained in below example:

Example:

Let the given expression be “2 3 1 * + 9 -“. We scan all elements one by one.

- 1) Scan ‘2’, it’s a number, so push it to stack. Stack contains ‘2’
- 2) Scan ‘3’, again a number, push it to stack, stack now contains ‘2 3’ (from bottom to top)
- 3) Scan ‘1’, again a number, push it to stack, stack now contains ‘2 3 1’
- 4) Scan ‘*’, it’s an operator, pop two operands from stack, apply the * operator on operands, we get $3*1$ which results in 3. We push the result ‘3’ to stack. Stack now becomes ‘2 3’.
- 5) Scan ‘+’, it’s an operator, pop two operands from stack, apply the + operator on operands, we get $3 + 2$ which results in 5. We push the result ‘5’ to stack. Stack now becomes ‘5’.
- 6) Scan ‘9’, it’s a number, we push it to the stack. Stack now becomes ‘5 9’.
- 7) Scan ‘-’, it’s an operator, pop two operands from stack, apply the – operator on operands, we get $5 - 9$ which results in -4. We push the result ‘-4’ to stack. Stack now becomes ‘-4’.

Program:

```
#include<stdio.h>                                //preprocessor directives
void sum();
void diff();                                    //function declaration
void mult();
void div();
void power();
int stack[50],top=-1;                          //initializing stack and top as -1
int main()
{
    char st[30];                                //array declaration
    int i;
    printf("enter the postfix expression\n");    //prints input
    scanf("%s",st);
    for(i=0;st[i]!='\0';i++)                    //for loop to evaluate expression
    {
        if(st[i]!=' ')                        //character shouldn't be space
        {
            switch(st[i])    //switch condition to push operators and operands
            {
                case '+':
                    sum();    //sum function call
                    break;    //loop terminate
                case '-':
                    diff();    //diff function call
                    break;
                case '*':
```

```

        mult();                //mult function call
        break;
    case '/':
        div();                //div function call
        break;
    case '^':
        power();              //power function call
        break;
    default:
        top++;                //top increments by 1
        stack[top]=st[i]-48;    //operand gets stored in top
    }
}
}
printf("the result is=%d",stack[top]);    //prints result
}

void sum()                    //sum called function
{
    int res,op1,op2;
    op1=stack[top];          //pops top element
    top--;                    //top decrements by 1
    op2=stack[top];          //pops top element
    top--;                    //top decrements by 1
    res=op2+op1;              //performs addition
    top++;                    //top incrementation
    stack[top]=res;           //result at top
}

void diff()                   //diff called function
{

```

```

int res,op1,op2;
op1=stack[top];           //pops top element
top--;                     //top decrementation
op2=stack[top];
top--;
res=op2-op1;               //performs difference
top++;
stack[top]=res;            //result at top
}

void mult()                //mult called function
{
    int res,op1,op2;
    op1=stack[top];        //pops top element
    top--;                 //top decrementation
    op2=stack[top];
    top--;
    res=op2*op1;           //performs multiplication
    top++;
    stack[top]=res;        //result at top
}

void div()                 //div called function
{
    int res,op1,op2;
    op1=stack[top];        //pops top element
    top--;                 //top decrementation
    op2=stack[top];
    top--;
    res=op2/op1;           //performs division
    top++;

```

```

stack[top]=res;                //result at top
}
void power()                   //power called function
{
    int res=1,op1,op2,i;
    op1=stack[top];            //pops top element
    top--;
    op2=stack[top];
    top--;
    for(i=0;i<op1;i++)          //loop
    {
        res=res*op2;            //finds power
    }
    top++;
    stack[top]=res;             //result at top
}

```

Algorithm:

Step 1: If a character is an operand push it to Stack

Step 2: If the character is an operator

Pop two elements from the Stack.

Operate on these elements according to the operator, and push the result back to the Stack

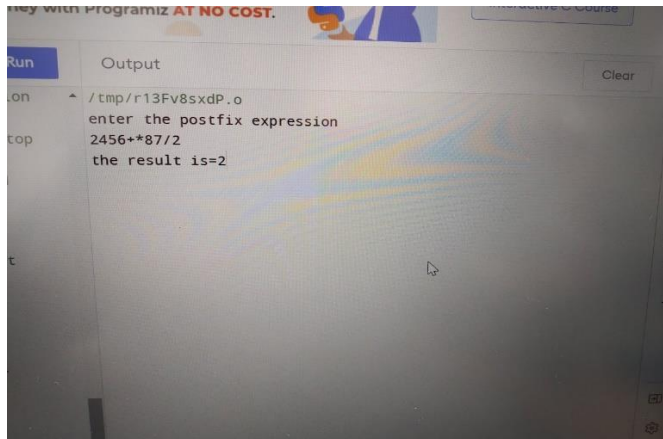
Step 3: Step 1 and 2 will be repeated until the end has reached.

Step 4: The Result is stored at the top of the Stack,

return it.

Step 5: End

Output:



A screenshot of a terminal window. At the top, there is a banner that says "Key with Programiz AT NO COST." followed by a logo. Below the banner, the terminal shows the following text: "enter the postfix expression", "2456**87/2", and "the result is=2". The terminal window has a "Run" button on the left and a "Clear" button on the right. The background of the terminal is dark with light-colored text.

```
Run Output Clear
/tmp/r13Fv8sxdP.o
enter the postfix expression
2456**87/2
the result is=2
```

PREFIX EVALUATION

Aim: To evaluate a prefix expression using stack.

Theory: In prefix notation, the operators come before their operands. It is also called Polish notation, or Warsaw notation. Below is the same equation in prefix notation:- $* + 2 2 3 10$

For the evaluation of prefix notation, we also use the stack data structure.

The following are the rules for evaluating prefix notation using a queue:

- Reverse the given expression.
- Start scanning from left to right.
- If the current value is an operand, push it onto the stack.
- If the current is an operator, pop two elements from the stack, apply the at-hand operator on those two popped operands, and push the result onto the stack.
- At the end, pop an element from the stack, and that is the answer.

Program:

```
#include<stdio.h>                //preprocessor directives

#include<string.h>

int top=-1,stack[30];            //initializing top as -1

void sum();                      //function declaration
void diff();                    //function declaration
void mult();                    //function declaration
void div();                     //function declaration
void power();                   //function declaration
```

```

int main()                                //main function
{
    int i;                                //initializing i

    char st[30];

    printf("Enter the prefix expression:");    //prints output

    scanf("%s",st);

    for(i=strlen(st)-1;i>=0;i--)            //for loop
    {
        if(st[i]!=' ')                    //to execute spaces
        {
            switch(st[i])                //switch condition
            {
                case '+':sum();            //function call for sum
                    break;                //terminates loop

                case '-':diff();           //function call for diff
                    break;

                case '*':mult();           //function call for mult
                    break;

                case '/':div();            //function call for div
                    break;

                case '^':power();          //function call for power
                    break;

                default:top++;              //increments top by 1

                stack[top]=st[i]-48;        //operand at stack top
            }
        }
    }
}

```



```

    }

}

}

printf("The result is:%d",stack[top]);          //prints result
}

void sum()                                     //called function for sum
{
    int res,op1,op2;                          //initializing variables for res,op1 and op2
    op1=stack[top];                           //returns stack[top]
    top--;                                    //top decrementation
    op2=stack[top];                           //returns stack[top]
    top--;                                    //top decrementation
    res=op1+op2;                              //performs addition
    top++;                                    //top incrementation
    stack[top]=res;                           //result at stack[top]
}

void diff()                                   //called function for diff
{
    int res,op1,op2;                          //initializing res op1 and op2
    op1=stack[top];                           //pops top element
    top--;                                    //top decrementation
    op2=stack[top];                           //returns stack[top]
    top--;
    res=op1-op2;                              //performs subtraction

```

```
    top++;                                //top incrementation
    stack[top]=res;                        //result at top
}
```

```
void mult()                               //called function for mult
```

```
{
    int res,op1,op2;
    op1=stack[top];                       //pops top element
    top--;                                //top decrementation
    op2=stack[top];
    top--;
    res=op1*op2;                           //performs multiplication
    top++;                                //incrementing top by 1
    stack[top]=res;                        //result at top
}
```

```
void div()                                //called function for div
```

```
{
    int res,op1,op2;
    op1=stack[top];                       //pops top element
    top--;                                //top decrementation
    op2=stack[top];
    top--;
    res=op1/op2;                           //performs division
    top++;                                //top incrementation
    stack[top]=res;                        //result at top
}
```

```

}

void power()                //called function for power
{
    int i,op1,op2,res=1;

    op1=stack[top];         //pops top element
    top--;                  //top decrementation
    op2=stack[top];
    top--;
    for(i=0;i<op2;i++)      //for loop to find power
    {
        res=res*op2;        //finds power
    }
    top++;                  //top incrementation
    stack[top]=res;         //result at top
}

```

Algorithm:

Step 1: Put a pointer P at the end of the end

Step 2: If character at P is an operand push it to Stack

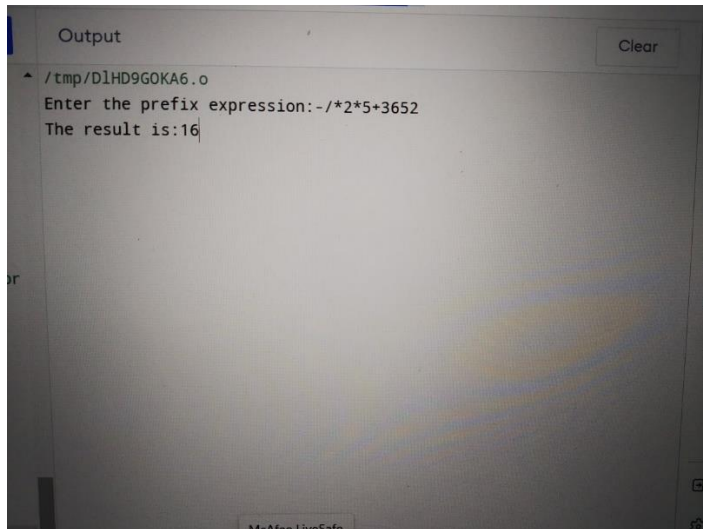
Step 3: If the character at P is an operator pop two
elements from the Stack. Operate on these elements
according to the operator, and push the result
back to the Stack

Step 4: Decrement P by 1 and go to Step 2 as long as there
are characters left to be scanned in the expression.

Step 5: The Result is stored at the top of the Stack,
return it

Step 6: End

Output:



```
Output
^ /tmp/D1HD9GOKA6.o
Enter the prefix expression:-/*2*5+3652
The result is:16
```

McAfee LiveSafe

INFIX EVALUATION

Aim: To evaluate an infix expression using stack.

Theory: We will take two stacks one for operator and another for operand.

Now while scanning the expression, as soon as we get an operand we push that in the operand stack.

If we get an opening bracket while scanning the expression, we push that in the operator stack.

If we get a closing bracket while scanning the expression, we pop the items of the operator stack until we get an opening bracket. And as soon as we get an opening bracket we pop that out too.

If an operator comes then all the operators in the operator stack with greater or equal precedence gets pop out until we get an opening bracket or the operator stack empties out. And then we push our current operator.

And whenever an operator is popped out then at the same time two elements from the operand stack are also popped out and an operation is performed using two operands and one operator. While performing this operation, operand that is popped out second will be placed first and operand which is popped out first will be placed second i.e. after operator.

Then the solved value needs to be pushed into the operand stack.

Program:

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <ctype.h>
```

```
#include <string.h>
```

```
//global variables  
int numbers[100], tn=-1, to=-1;  
char op[100];
```

```
//is it digit?
```

```
int isItDigit(char c)
```

```
{  
    switch(c)  
    {  
        case '1':  
            return 1;  
        case '2':  
            return 2;  
        case '3':  
            return 3;  
        case '4':  
            return 4;  
        case '5':  
            return 5;  
        case '6':  
            return 6;  
        case '7':  
            return 7;  
        case '8':  
            return 8;  
        case '9':  
            return 9;  
    }
```

```

    return 0;
}
//function to push digits
void pushNum(int n)
{
    numbers[++tn]=n;
}
//function to push operators
void pushOp(char ch)
{
    op[++to]=ch;
}
//function to pop digits
int popNum()
{
    return numbers[--tn];
}
//function to pop operators
char popOp()
{
    return op[--to];
}
//actual operation
int actualOperation(int numbers[50], char op[50])
{
    int x,y;
    char opr;
    x=popNum();

```

```

y=popNum();
opr=popOp();
    switch(opr)
    {
        case '+':
            return x+y;
        case '-':
            return y-x;
        case '*':
            return x*y;
        case '/':
            if(x==0)
            {
                printf("\nCannot divide by zero");
                exit(0);
            }
            else
            {
                return y/x;
            }
        }
    return 0;
}

//function to check if character is an operator or not
int isOperator(char ch)
{
    switch(ch)
    {

```



```

        case '+':
            return '+';
        case '-':
            return '-';
        case '*':
            return '*';
        case '/':
            return '/';
    }
}

//precedence of the operators
int precedence(char ch)
{
    switch (ch)
    {
        case '+':
            return 1;
        case '-':
            return 1;
        case '*':
            return 2;
        case '/':
            return 2;
        case '^':
            return 3;
    }

    return -1;
}

```

```

}
//to evaluate an infix expression
int evaluateInfix(char expr[50])
{
    int i, num, output, r;
    char c;

    for(i=0;expr[i]!=0;i++)
    {
        c = expr[i];

        if(isltDigit(c)!=0)
        {
            num = 0;
            while(isltDigit(c))
            {
                num = isltDigit(c);
                i++;
                if(i < strlen(expr))
                {
                    c = expr[i];
                }
                else
                {
                    break;
                }
            }
            i--;
            pushNum(num);
        }
    }
}

```

```

else if(c=='(')
{
    pushOp(c);
}
else if(c==')')
{
    while(op[to++]!='(')
    {
        r = actualOperation(numbers, op);
        pushNum(r);
    }
    popOp();
}
else if(isOperator(c))
{
    while(to!=-1 && precedence(c)<=precedence(op[to]))
    {
        output = actualOperation(numbers, op);
        pushNum(output);
    }
    pushOp(c);
}
}
while(to!=-1)
{
    output = actualOperation(numbers, op);
    pushNum(output);
}

```

```

    return popNum();
}
//main function
int main()
{
    char expr[50]
    printf("enter the infix expression");
    gets(expr)
    int ans = evaluateInfix(expr);
    printf("Answer = %d", ans);
}

```

Algorithm:

Step 1: Pop out two values from the operand stack, lets say it is A and B.

Step 2: Pop out operation from operator stack. Lets say it is '+'

Step 3: Perform A+B and push the result to the operand stack.

Output:

