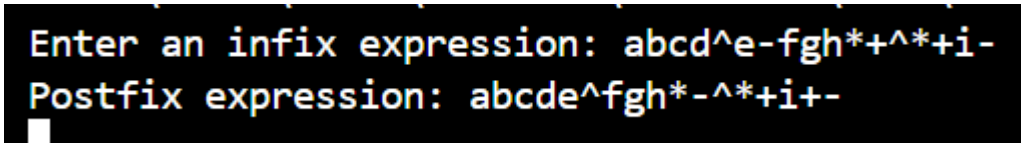
Program 2

Write A Program to convert a given valid parenthesized infix arithmetic expression to postfix expression. The expression consists of single character operands and the binary operators + (plus), - (minus), * (multiply) and / (divide)

Code:

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

#define MAX 5

char stack[MAX];
int top = -1;

void push(char c) {
   if (top < MAX - 1) {
      stack[++top] = c;
   }
}
char pop() {
   if (top >= 0) {
      return stack[top--];
   }
   return '\0';
}
char peek() {
   if (top >= 0) {
      return stack[top];
   }
   return '\0';
}

int precedence(char c) {
   switch (c) {
      case '+': return 1;
      case '-': return 1;
      case '*': return 2;
      case '/': return 2;
      case '^': return 3;
      default: return 0;
   }
}


int isOperator(char c) {
   return c == '+' || c == '-' || c == '*' || c == '/'|| c=='^';
```

```
}


void infixToPostfix(const char *infix, char *postfix) {
    int i = 0, j = 0;
    while (infix[i]) {
        if (isalnum(infix[i])) {
            postfix[j++] = infix[i];
        } else if (infix[i] == '(') {
            push(infix[i]);
        } else if (infix[i] == ')') {
            while (top != -1 && peek() != '(') {
                postfix[j++] = pop();
            }
            pop();
        } else if (isOperator(infix[i])) {
            while (top != -1 && precedence(peek()) >= precedence(infix[i])) {
                postfix[j++] = pop();
            }
            push(infix[i]);
        }
        i++;
    }
    while (top != -1) {
        postfix[j++] = pop();
    }
    postfix[j] = '\0';
}

int main() {
    char infix[MAX], postfix[MAX];

    printf("Enter an infix expression: ");
    scanf("%s", infix);

    infixToPostfix(infix, postfix);
    printf("Postfix expression: %s\n", postfix);

    return 0;
}
```



7/10

Q8. WAP to convert a given valid parentheses infix arithmetic expression to postfix expression.

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

int prec(char c) {
    if (c == '^')
        return 3;
    else if (c == '/' || c == '*')
        return 2;
    else if (c == '+' || c == '-')
        return 1;
    else
        return -1; }

char associativity(char c) {
    if (c == '^')
        return 'R';
    return 'L';
}

void infixToPostfix(const char *s) {
    int len = strlen(s);

char *result = (char *) malloc (len+1);
char *stack = (char *) malloc (len);
int resultIndex = 0;
int stackIndex = -1;
```

```
if (!result || !stack) {
    printf("Memory allocation failed!\n");
    return;
}

for(int i = 0; i < len; i++) {
    char c = s[i];

if((c>='a' && c<='z') || (c>='A' && c<='Z')
    || (c>='0' && c<='9')) {
        result[resultIndex++] = c;
}
else if (c == '(') {
    stack[++stackIndex] = c;
}
else if (c == ')') {
    while (stackIndex >= 0 && stack[stackIndex]
!= '(') {
        result[resultIndex++] = stack[stackIndex--];
    }
    stackIndex--;
}
else {
    while(stackIndex >= 0 && (prec(c) <
prec(stack[stackIndex]) || (prec(c)== prec
(stack[stackIndex]) && associativity(c)
== 'L')))
    {
        result[resultIndex++] = stack[stackIndex--];
}
```

```
stack[++stackIndex] = c;
}
}
while(stackIndex >= 0) {
    result[resultIndex++] = stack[stackIndex--];
}
result[resultIndex] = '\0';
printf("%s\n", result);

free(result);
free(stack); }

int main() {
    char exp[] = "a+b*(c^d-e)^(f+g*h)
                  -i";
                  user input
infixToPostfix(exp);
return 0;
}
```

output

abcd^e-fgh*+^*+i-

Namaska M
7/10/2024