Week 2

Q) WAP 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), * (multiple) and / (divide).

pseudocode:-

```
Algorithm Infixtopostfix(exp){:
    initialize string, stack[];
    if token == operand:
        print token
    else if token == '(':
        push token
    else if token == ')':
        pop stack till '(' encountered;
        pop '(';
    else if token == operator
        if stack is empty:
            push operator
        if stack is not empty:
            compare precedence of
            stack[top] and token
                if token > stack[top] (precedence
                    push token;
                else if stack[top] > token:
                    pop ~~print~~ stack[top] ~~ttt~~
                    and compare next top
```

else if stack[top] == token (operator):
use associativity rule
~~print~~
End of Algorithm;

#CODE#

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#define MAX 100
char stack[MAX]
int top = -1;
void push(char c){
  stack[++top] = c;
  }
char pop(){
    return stack[top--];
  }
int precedence(char op){
  if(op == '+' || op == '-')
      return 1;
  if(op == '*' || op == '/')
      return 2
  if(op == '^')
      return 3
} return 0;
```

```
int associativity (char op){
    if (op == '^')
        return 1;
    return 0;
}
void infixToPostfix (char* infix, char* postfix){
    int i, j=0;
    char token;
    for (i=0; i<strlen(infix); i++){
        token = infix[i];
        if (isalnum(token)){
            postfix[j++] = token;
        } else if (token == '('){
            push(token);
        } else if (token == ')'){
            while (top != -1 && stack[top] != '('){
                postfix[j++] = pop();
            }
            if (top != 1)
                pop();
```

```
} else if (token == '+' || token == '-' ||
    token == '*' || token == '/' ||
    token == '^') {
        while (top != -1 && stack[top] != '('
                && (precedence(stack[top])
                    > precedence(token) ||
                (precedence(stack[top]) ==
                precedence(token) && associativity
                (token) == 0))) {
            postfix[j++] = pop();
        }
        push(token);
    }
}
while(top != -1) {
    postfix[j++] = pop();
}
    postfix[j] = '\0';
}
```

```
int main()
{
    char infix[MAX], postfix[MAX];
    printf("Enter infix : ");
    fgets(infix, MAX, stdin);
    infixTOpostfix(infix, postfix);
    printf("Postfix expression : %s\n",
                postfix);
    return 0;
}
```

OUTPUT

Enter infix : (A+B)*C+(D^M)+(E-F).

Postfix expression: AB+C*DM^+EF-+.