Algorithm to convert from Infix notation to Postfix notation

Data structure used:
stack

Input:
string **infix**;
Integers are single digits.

Output:
string **postfix**;

Algorithm:

1. Push a left parenthesis '(' onto the stack;
2. Append a right parenthesis ')' to the end of **infix**.
3. While the stack is not empty, read **infix** from left to right and do the following:
   a) If the current character in **infix** is a white space, simply ignore it.
   b) If the current character in **infix** is a digit, copy it to the next element of **postfix**.
   c) If the current character in **infix** is a left parenthesis, push it onto the stack.
   d) If the current character in **infix** is an operator,
      - Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and insert the popped operators in **postfix**.
      - Push the current character in **infix** onto the stack.
   e) If the current character in **infix** is a right parenthesis,
      - Pop operators from the top of the stack and insert them in **postfix** until a left parenthesis is at the top of the stack.
      - Pop (and discard) the left parenthesis from the stack.

The following arithmetic operations are allowed in an expression:
+, –, *, /, ^ exponentiation, % modulus

Example:

Infix string	Postfix string	Value
(6+2) * 5 – 8 / 4	6 2 + 5 * 8 4 / –	38
2+3*4+5-6	234*+5+6-	13

Notice that parenthesis are not needed in the postfix string.

Routines you need:

- Function **convertToPostfix** that converts the infix expression to postfix notation.
- Function **isOperator** that determines if c is an operator.
- Function **precedence** that determines if the precedence of **operator1** is less than, equal to or greater than the precedence of **operator2**. The function returns –1, 0 and 1, respectively.

Algorithm to evaluate a Postfix notation

Algorithm:

1. Append the null character '\0' to the end of the **postfix** expression. When the null character is encountered, no further processing is necessary.
2. While '\0' has not been encountered, read the **postfix** expression from left to right.
   a) If the current character in **infix** is a white space, simply ignore it.
   b) If the current character is a digit,
      - Push its integer value onto the stack (the integer value of a digit character is its ASCII value minus the ASCII value of zero).
   c) Otherwise, if the current character is an *operator* (one of +, –, *, /, ^ exponentiation, or % modulus),
      - Pop the two top elements off the stack into variables ***x*** and ***y***.
      - Calculate ***y operator x***.
      - Push the result of the calculation onto the stack.
3. When the null character '\0' is encountered in the expression, pop the top value off the stack. This is the result of the **postfix** expression.

Routines you need:

- Function **evaluatePostfixExpression** that evaluates the postfix expression.
- Function **calculate** that evaluates the expression **op1 operator op2**.

Routines for the stack:

- Function **push** and **pop** for the stack.
- Function **stackTop** that returns the top value of the stack without popping the stack.
- Function **isEmpty** that determines if the stack is empty.
- Function **printStack** that prints the stack.