CS2400 Fall 2018 Project 2

Total points: 100

Due date: Wednesday, October 10, 2018

Purpose:

1. Using the ADT stack to process algebraic expressions
2. Implement the ADT stack by using an array (resizable), a linked chain, and a vector

Task Descriptions:

Task 1: Show the contents of the stack as you trace the algorithm *checkBalance*, given in Segment 5.8 (or see the algorithm *checkBalance* in this pdf), for the following expression:

- $\{a\ (\ b\ *\ c\ )\ /\ [d\ +\ e]\ /\ f\ )\ -\ g\}$

Task 2: Using the algorithm *convertToPostfix*, given in Segment 5.16 (or see the algorithm *convertToPostfix* in this pdf), convert the following infix expression to postfix expression:

- $(a\ -\ b\ *\ c)\ /\ (d\ *\ e\ *\ f\ +\ g)$

Task 3: Using the algorithm *evaluatePostfix*, given in Segment 5.18 (or see the algorithm *evaluatePostfix* in this pdf), evaluate the following postfix expression. Assume that a = 2, b = 3, c = 4, d = 5, and e = 6.

- $a\ b\ *\ c\ a\ -\ /\ d\ e\ *\ +$

Task 4: Show the contents of the two stacks as you trace the algorithm *evaluateInfix*, given in Segment 5.21 (or see the algorithm *evaluateInfix* in this pdf), to evaluate the following infix expression. Assume that a = 2, b = 3, c = 4, d = 5, e = 6, and f = 7.

- $(d\ *\ f\ +\ 1)\ *\ e\ /\ (a\ \hat{}\ b\ -\ b\ *\ c\ +\ 1)\ -\ 72$

Task 5: Redesign your previous implementations (a resizable array and a linked chain) of project 1 for our project 2, following the interface of stack.

Task 6: Repeat Task 5, but use a vector.

What to Submit?

1. Written document for Tasks 1-4.
2. Source codes for Tasks 5-6.
3. Please zip all documents as yourname_p2.zip and submit it in blackboard.

You will be graded based on the quality of your program and the correctness of your algorithm analysis.

List of Algorithms (as Shown in Our Textbook)

1. The algorithm *checkBalance*

```
Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets, and braces in an expression are paired correctly.

isBalanced = true
while ((isBalanced == true) and not at end of expression)
{
   nextCharacter = next character in expression
   switch (nextCharacter)
   {
      case '(': case '[': case '{':
         Push nextCharacter onto stack
         break

      case ')': case ']': case '}':
         if (stack is empty)
            isBalanced = false
         else
         {
            openDelimiter = top entry of stack
            Pop stack
            isBalanced = true or false according to whether openDelimiter and
                         nextCharacter are a pair of delimiters
         }
         break
   }
}

if (stack is not empty)
   isBalanced = false
return isBalanced
```

2. The algorithm *convertToPostfix*

```
Algorithm convertToPostfix(infix)
// Converts an infix expression to an equivalent postfix expression.

operatorStack = a new empty stack
postfix = a new empty string
while (infix has characters left to parse)
{
   nextCharacter = next nonblank character of infix
   switch (nextCharacter)
   {
      case variable:
         Append nextCharacter to postfix
         break

      case '^' :
         operatorStack.push(nextCharacter)
         break

      case '+' : case '-' : case '*' : case '/' :

         while (!operatorStack.isEmpty() and
                precedence of nextCharacter <= precedence of operatorStack.peek())
         {
            Append operatorStack.peek() to postfix
            operatorStack.pop()
         }
         operatorStack.push(nextCharacter)
         break

      case '( ' :
         operatorStack.push(nextCharacter)
         break

      case ')' : // stack is not empty if infix expression is valid
         topOperator = operatorStack.pop()
         while (topOperator != '(')
         {
            Append topOperator to postfix
            topOperator = operatorStack.pop()
         }
         break

      default: break
   }
}

while (!operatorStack.isEmpty())
{
   topOperator = operatorStack.pop()
   Append topOperator to postfix
}
return postfix
```

3. The algorithm *evaluatePostfix*

```
Algorithm evaluatePostfix(postfix)
// Evaluates a postfix expression.

valueStack = a new empty stack
while (postfix has characters left to parse)
{
   nextCharacter = next nonblank character of postfix
   switch (nextCharacter)
   {
      case variable:
         valueStack.push(value of the variable nextCharacter)
         break

      case '+' : case '-' : case '*' : case '/' : case '^' :
         operandTwo = valueStack.pop()
         operandOne = valueStack.pop()
         result = the result of the operation in nextCharacter and its operands
                  operandOne and operandTwo
         valueStack.push(result)
         break

      default: break
   }
}

return valueStack.peek()
```

4. The algorithm *evaluateInfix*

```
Algorithm evaluateInfix(infix)
// Evaluates an infix expression.

operatorStack = a new empty stack
valueStack = a new empty stack
while (infix has characters left to process)
{
   nextCharacter = next nonblank character of infix
   switch (nextCharacter)
   {
      case variable:
         valueStack.push(value of the variable nextCharacter)
         break

      case '^' :
         operatorStack.push(nextCharacter)
         break

      case '+' : case '-' : case '*' : case '/' :
         while (!operatorStack.isEmpty() and
               precedence of nextCharacter <= precedence of operatorStack.peek())
         {
            // Execute operator at top of operatorStack
            topOperator = operatorStack.pop()
            operandTwo = valueStack.pop()
            operandOne = valueStack.pop()
            result = the result of the operation in topOperator and its operands
                     operandOne and operandTwo
            valueStack.push(result)
         }
         operatorStack.push(nextCharacter)
         break

       case '(' :
          operatorStack.push(nextCharacter)
          break

       case ')' : // stack is not empty if infix expression is valid
          topOperator = operatorStack.pop()
          while (topOperator != '(')
          {
             operandTwo = valueStack.pop()
             operandOne = valueStack.pop()
             result = the result of the operation in topOperator and its operands
                      operandOne and operandTwo
             valueStack.push(result)
             topOperator = operatorStack.pop()
          }
          break

        default: break
   }
}

while (!operatorStack.isEmpty())
{
   topOperator = operatorStack.pop()
   operandTwo = valueStack.pop()
   operandOne = valueStack.pop()
   result = the result of the operation in topOperator and its operands
            operandOne and operandTwo
   valueStack.push(result)
}
return valueStack.peek()
```