Ron Cox

HW2
605.202.81
Data Structures

1. Stack Operations

a) Set `i` to the bottom element of the stack

```
procedure getBottomElement(stack)
      auxiliaryStack = createEmptyStack()

      while not stack.empty()
      auxiliaryStack.push(stack.pop())

      i = auxiliaryStack.peek()  // bottom element

      while not auxiliaryStack.empty()
      stack.push(auxiliaryStack.pop())

      return i
end procedure
```

b) Set `i` to the third element from the bottom of the stack

```
procedure getThirdFromBottom(stack)
      auxiliaryStack = createEmptyStack()

      while not stack.empty()
      auxiliaryStack.push(stack.pop())

      count = 0
      thirdFromBottom = null

      while not auxiliaryStack.empty()
      count = count + 1
      if count == 3
            thirdFromBottom = auxiliaryStack.peek()
      stack.push(auxiliaryStack.pop())
```

```
    i = thirdFromBottom
    return i
end procedure
```

2. Checking Delimiters with Stack

a. `{[A+B]-[(C-D)]`

```
// Initial Stack: []
push '{'    // Stack: ['{']
push '['    // Stack: ['{', '[']
push ']'    // Stack: ['{']
push '-'    // Stack: ['{', '-']
push '['    // Stack: ['{', '-', '[']
push '('    // Stack: ['{', '-', '[', '(']
push ')'    // Stack: ['{', '-', '[']
push ']'    // Stack: ['{', '-']
push '}'    // Stack: []
```

b. `((H) * {([J+K])})`

```
// Initial Stack: []
push '('    // Stack: ['(']
push '('    // Stack: ['(', '(']
push ')'    // Stack: ['(']
push '*'    // Stack: ['(', '*']
push '{'    // Stack: ['(', '*', '{']
push '('    // Stack: ['(', '*', '{', '(']
push '['    // Stack: ['(', '*', '{', '(', '[']
push ')'    // Stack: ['(', '*', '{', '(']
push ']'    // Stack: ['(', '*', '{']
push ')'    // Stack: ['(', '*']
push '}'    // Stack: ['(']
push ')'    // Stack: []
```

3. Determine if String is of the Form xCy

```
procedure isFormXCX(inputString)
     stack = createEmptyStack()

     for each char in inputString
     if char == 'C'
          break
     stack.push(char)

     for each char in inputString (continuing from 'C')
     if char != stack.pop()
          return false

     return stack.empty()
end procedure
```

4. Determine if String is of the Form aDbDc...Dz

```
procedure isFormADBCD(inputString)
     stack = createEmptyStack()
     isValid = true

     for each char in inputString
     if char == 'C'
          while not stack.empty()
               if stack.pop() != stack.pop()
                    isValid = false
                    break
          if not isValid
               return false
     else if char == 'D'
          if stack.empty() or not isValid
               return false
          stack = createEmptyStack()
          isValid = true
     else
```

```
            stack.push(char)

        return stack.empty() and isValid
end procedure
```

5. Implement a One-Dimensional Array Using Two Stacks

```
// Assume the stacks are s1 and s2 for storage
procedure arrayInsert(index, value)
        while s1 not empty
        s2.push(s1.pop())

        s1.push(value)

        while s2 not empty
        s1.push(s2.pop())
end procedure

procedure arrayRead(index)
        while s1 not empty and index > 0
        s2.push(s1.pop())
        index = index - 1

        value = s1.peek()

        while s2 not empty
        s1.push(s2.pop())

        return value
end procedure
```

6. Two Stacks in a Single Array

```
array = new array[SPACESIZE]
top1 = -1
top2 = SPACESIZE

procedure push1(value)
      if top1 < top2 - 1
      top1 = top1 + 1
      array[top1] = value
      else
      error "Stack Overflow"
end procedure

procedure push2(value)
      if top1 < top2 - 1
      top2 = top2 - 1
      array[top2] = value
      else
      error "Stack Overflow"
end procedure

procedure pop1()
      if top1 >= 0
      value = array[top1]
      top1 = top1 - 1
      return value
      else
      error "Stack Underflow"
end procedure

procedure pop2()
      if top2 < SPACESIZE
      value = array[top2]
      top2 = top2 + 1
      return value
      else
      error "Stack Underflow"
end procedure
```

7. Transform Expressions

a) `(A + B) * (C $ (D - E) + F) - G`

- Prefix: `- * + A B + C $ - D E F G`
- Postfix: `A B + C D E - $ F + * G -`

b) `A + ((B - C) * (D - E) + F) / G) $ (H - J)`

- Prefix: `$ + A / + * - B C - D E F G - H J`
- Postfix: `A B C - D E - * F + G / H J - $`

8. Transform Prefix Expressions to Infix

a) `++A - * $ B C D / + E F * G H I`

- Infix: `A + ((B $ C) * D - (E + F / (G * H)) + I)`

b) `+ - $ A B C * D * * E F G`

- Infix: `((A $ B - C) + ((D * E) * (F * G)))`

c) `A B - C + D E F - + $`

- Infix: `((A - B) + C) $ (D + (E - F))`

d) `A B C D E - + $ * E F * -`

- Infix: `((A - ((B $ (C + D)) * E)) * (E - F))`

9. Evaluate Postfix Expressions

a) `A B + C - B A + C $ -`

- Stack operations:
    1. Push A (1)
    2. Push B (2)
    3. Add (1 + 2 = 3)
    4. Push C (3)
    5. Subtract (3 - 3 = 0)
    6. Push B (2)
    7. Push A (1)
    8. Add (2 + 1 = 3)
    9. Push C (3)
    10. Exponentiate (3 $ 3 = 27)
    11. Subtract (0 - 27 = -27)

- Result: -27

10. Prefix Function for Infix to Prefix Conversion

```
procedure infixToPrefix(infix)
     stack = createEmptyStack()
     prefix = createEmptyString()

     for each char in infix (right to left)
     if char is operand
          prefix.prepend(char)
     else if char is ')'
          stack.push(char)
     else if char is '('
          while stack.peek() != ')'
               prefix.prepend(stack.pop())
          stack.pop() // remove ')'
     else
          while not stack.empty() and precedence(char) <
precedence(stack.peek())
               prefix.prepend(stack.pop())
          stack.push(char)

     while not stack.empty()
     prefix.prepend(stack.pop())

     return prefix
end procedure
```