# Correction of the Tutorial Series N°4

**Exercise 1:**

Given a stack **S** of integers, the following questions are independent:

1. Write the procedure **nbEvenOdd(S, nbEven, nbOdd)** that returns the number of even and odd elements in a stack **S** passed as parameters. At the end, stack **S** should be empty.
2. Write the procedure **reorganize(S)** that reorganizes the stack **S** passed as a parameter so that negative elements are at the bottom and positive elements are at the top. Note: It is not required to sort the stack.
3. Write a function named **reverse_stack(S)** that takes a stack **S** as a parameter and returns a new stack containing the elements of **S** in reverse order. The original stack **S** should remain unchanged. Indication: Use two stacks.

**Solution:**

Let's begin with the declaration of the data structure to use:

```
Type Stack = ^Node;
     Node = Record
          Begin
          Val: integer;
          Next: Stack;
          End;
```

1.

```
Procedure nbEvenOdd(Var S: Stack; Var nbEven, nbOdd: Integer);
Var x: Integer;
Begin
nbEven ← 0;
nbOdd ← 0;
While isStackEmpty(S) = False Do
     Begin
     pop(x, S);
     If x mod 2 = 0 Then nbEven ← nbEven + 1
     Else nbOdd ← nbOdd + 1;
     End;
End;
```

2.

```
Procedure reorganize(Var S: Stack);
Var SPos, SNeg: Stack; e: Integer;
Begin
initializeStack(SPos);
initializeStack(SNeg);
While isStackEmpty(S) = False Do
     Begin
     pop(e, S);
     If e > 0 Then push(e, SPos)
     Else push(e, SNeg);
```

```
        End;
    While isStackEmpty(SNeg) = False Do
            Begin
            pop(e, SNeg);
            push(e, S);
            End;
    While isStackEmpty(SPos) = False Do
            Begin
            pop(e, SPos);
            push(e, S);
            End;
    End;
```

3.

```
    Function reverse_Stack(S: Stack): Stack;
    Var S2, S3: Stack; x: Integer;
    Begin
    initializeStack(S2);
    initializeStack(S3);
    While isStackEmpty(S) = False Do
        Begin
        pop(x, S);
        push(x, S3);
        push(x, S2);
        End;
    While isStackEmpty(S3) = False Do
        Begin
        pop(x, S3);
        push(x, S2);
        End;
    Reverse_Stack ← S2;
    End;
```

**Exercise 2 :**

An arithmetic expression can contain digits, symbols, mathematical operators, and parentheses. Parentheses can be nested within each other, for example:

$$f(x) = \left(\left(x^2 + 2x + 3\right)/\left(\left(4x + 2\right)*3\right)\right)$$

Write a function that takes an arithmetic expression (in the form of a string) as a parameter and determines whether it is correctly parenthesized, meaning all parentheses are properly closed.

***Indication:*** The expected solution involves using a stack.

**Solution:**

```
    Type Stack = ^Node;
        Node = Record
            Begin
            Val: Character;
            Next: Stack;
            End;
Function wellParenthesized(exp: String[50]): Boolean;
Var S: Stack; i: Integer; correct: Boolean; x: Character;
Begin
initializeStack(S);
```

```
i ← 1;
correct ← True;
While i ≤ Length(exp)) AND correct=True Do
     Begin
     If exp[i] = '(' Then push('(', S)
     Else if exp[i] = ')' Then
          Begin
          If isStackEmpty(S) = False Then
                pop(x, S);
          Else correct ← False;
          End;
     i ← i + 1;
     End;
If isStackEmpty(S) = False Then correct ← False;
wellParenthesized ← correct;
End;
```

**Exercice 3:**

A palindrome is a string of characters that reads the same from left to right or from right to left (e.g., "level", "radar", "civic").

Using only one stack and one queue, write a function that tests whether a string (stored in a linked list of characters) is a palindrome or not.

**Solution :**

```
Type   Node = Record
            Begin
            val: character;
            Next: ^Node;
            End;
       List  = ^Node;
       Stack = ^Node;
       Queue = ^Node;
Function is_palindrome(L: List): Boolean;
Var S: Stack; Q: Queue; p: List; x, y: Character; pal: Boolean;
Begin
p ← L ;
initializeStack(S);
initializeQueue(Q);
While p ≠ Nil Do
       Begin
       enqueue(p^.val, Q);
       push(p^.val, S);
       p ← p^.Next;
       End;
pal ← True;
While isStackEmpty(S) = False AND pal = True Do
       Begin
       pop(x, S);
       dequeue(y, Q);
       If x ≠ y Then pal ← False;
       End;
is_palindrome ← pal;
End;
```

**Exercise 4 :**

An equipment rental company receives orders every day. Each order has a number and a status (processed or not). At the end of each day, the responsible agent needs to enter all orders (processed and unprocessed) in the order they were received and store them in a data structure.

1. What is the most suitable data structure for this problem? Provide the declaration of this structure.
2. Write a procedure that takes all orders as input and returns the number of processed and unprocessed orders.
3. Write a function that reorganizes the data structure containing all orders by placing processed orders at the end in the order of their arrival.
4. Modify the previous function so that processed orders are placed at the end in reverse order of their arrival.

**Solution :**

1. The most suitable data structure is **Queue**. The declaration is the following:

```
Type Order = Record
             Begin
             Num: Integer;
             State: Boolean;
             End;
      Queue = ^Node;
      Node = Record
             Begin
             val: Order;
             Next: Queue;
             End;
```

2.

```
Procedure nbOrders(q: Queue; Var nbProc, nbUnpro: Integer);
Var q2: Queue; o: Order;
Begin
initializeQueue(q2);
nbProc ← 0;
nbUnproc ← 0;
While isQueueEmpty(q) = False Do
       Begin
       dequeue(o, q);
       enqueue(o, q2);
       If o.State = True Then nbProc ← nbProc + 1
       Else nbUnproc ← nbUnproc + 1;
       End;
While isQueueEmpty(q2) = False Do
       Begin
       dequeue(o, q2);
       enqueue(o, q);
       End;
End;
```

3.

```
Procedure reorganize(q: Queue);
Var q2, q3: Queue; o: Order;
Begin
initializeQueue(q2);
initializeQueue(q3);
While isQueueEmpty(q) = False Do
```

```
        Begin
        dequeue(o, q);
        If o.State = False Then enqueue(o, q2)
        Else enqueue(o, q3);
        End;
While isQueueEmpty(q2) = False Do
        Begin
        dequeue(o, q2);
        enqueue(o, q);
        End;
While isQueueEmpty(q3) = False Do
        Begin
        dequeue(o, q3);
        enqueue(o, q);
        End;
End;
```

4.

```
Procedure reorganizeReverse(q: Queue);
Var q2: Queue; s: Stack; o: Order;
Begin
initializeQueue(q2);
initializeStack(s);
While isQueueEmpty(q) = False Do
        Begin
        dequeue(o, q);
        If o.State = False Then enqueue(o, q2)
        Else push(o, s);
        End;
While isQueueEmpty(q2) = False Do
        Begin
        dequeue(o, q2);
        enqueue(o, q);
        End;
While not isStackEmpty(s) = False Do
        Begin
        pop(o, s);
        enqueue(o, q);
        End;
End;
```