Programming Questions

a) Design and submit the pseudo-code for both the versions of arithmetic calculator.

First Version (using stack)

2 stacks, opStk (holds operators), valStk(holds values)

```
Algorithm doOp ()
No input and output
    op <- opStk.pop()
    if (op is unary operator (prec(op) == 2 or 3) then // unary case
        y <- valStk.pop()
        valStk.push( y op)
    else // binary case.
        x <- valStk.pop()
        y <- valStk.pop()
        valStk.push( y op x)
    // Also, for parenthesis case, I do not deal with this algorithm. Check EvalExp.
```

```
Algorithm prec(refOp) // Ranking algorithm for operators
Input : string of reference of operation
Output : integer of ranking of operation (lowest has more priority)
    If refOp is ( or ) then
        return 1
    else if refOp is ! then
        return 2
    else if refOp is unary – then
        return 3
    else if refOp is ^ then
        return 4
    else if refOp is * or / then
        reutrn 5
    else if refOp is + or binary – then
        return 6
    else if refOp is >, >=, <, or <= then
        return 7
    else if refOp is == or != then
        return 8
    else if refOp is $ then
        return 0 // this is when it rehearse to the last

Algorithm repeatOps(refOp) :
Input : String of reference of operation
Output : void
    if  prec(refOp) is 0 // if refOp is $ which means it reaches the end of given
expression (or end of parenthesis). Keep Doing operation to return final (parenthesis)
value
        while valStk.size() is greater than 1 and opStk is not "(" then
            doOp()
    while valStk.size() is greater than 1 AND prec(refOp) is not 1 AND prec(refOp) is
greater than or equal prec(opStk.top()) AND opStk.top() is not "(" then
        doOp(); // All case except parenthesis.
    if valStk size is 1 and (precc(refOp) is 2 or prec(refOp) is 3)
            doOp(); // unary case if valStk has one element.

Algorithm isNumber(s)
Input: String reference
Output: Boolean
    If s is a number then
```

```
        return true
    return false

Algorithm EvalExp(exp)
Input : String of arithmetic equation.
Output: Integer of calculated result of equation.
    valTemp // Temporary value
    for i = 0 to length of exp
        temp <- token of exp.
        if isNumber(temp) or “.”
            concatenated integer(or double) value to valTemp
        else
            if valTemp is not empty then
                 valStk.push(valTemp) // if valTemp is empty, nothing was
                concatenated for numbers so nothing to push to the stack
            else if temp is “(“
                creating new expression inside parenthesis
                valStk.push(EvalExp(new expression)) // In this way, calling
EvalExp again and it will accumulate existing stacks (It’s recursive call but it’s not based
on recursion, it’s still based on stack)
            else
                verifying operator (whether unary or not).// In the real code,
            there is full of if and else condition to verify.
                repeatOps(temp)
                opStk.push(temp)
    repeatOps(“$”)
    return valStk.pop()
```

Second version (recursion)

Using one arrays (Do not use any stack. It is just for keeping the token from equation). st(array). Using binary linked list (node) (for tree)

```
Static integer size

Algorithm EvalExp(expression)

Input: expression (equation)

Output: calculated or result value

tempN <- new Node <<indicating where is the node and it has left, right and prev
(previous) pointer>> //it can have only one parent since it's binary linked list

tempP <- tempN.left <<indicating where the tempNode is pointing>>

val <- temporary number value

while there is no more token

    token <- next token

    if token is number then

        val <- token

    else if prec(token) is 2 then // Unary ! case

        tempP(new Node(token))

        tempP.left(new Node(val))

    else if prec(token) is 3 then // Unary - case

        tempP(new Node(token))

        tempP.left(new Node(next token))

    else if binary operator then

        if (prec(token) >= prec(tempN))

            tempP(new Node(val)

            tempN = new Node(tempN.left(tempP) // It means previous operator
        should calculated first. In this way current operator will be parent of
        previous value.

        else

            tempP.right(new Node(val)
                T <- tempN
                tempN <- tempN.prev.right (new Node(token)
                tempN.left(T)
    else if token is “(“ then
        while token is not “)”
            token <- next token
            tempString += token
        tempP <- new Node(EvalExp(tempString)) // recursive call for parenthesis.
        If tempP is not right then // if parenthesis starts at first.
            tempP = tempN.right
tempP(val)
while nodeP.prev is note empty
    nodeP =nodeP.prev
return treeCalculation(nodeP)

Algorithm treeCalculation(nodeP)
Input: top (ancestor) of the tree
Output: calculated value
if nodeP is external // In other words, if there is no child.
    return nodeP.value()
else
    op <- nodeP.value() // if it’s not an external node, it should be operator
    if op is binary operator
        x <- treeCalculation(nodeP.left)
        y <- treeCalculation(nodeP.rgith)
        return x op y
    else
        x <- treeCalculation(nodeP.left)
        return x op
```

Explanation – Using same method of prec. Building up binary tree to calculate easily and depending of type of operator, the way it store is different as well. Similar to first version, whether ranking of operator is greater than or equal, or not, the way to assigning node is different.

b) Implement a Java program for the first version. In your program, you will implement and use your own array-based stack (and not the built-in Java stack) of variable size based on the doubling strategy discussed in class (refer to the code segments provided on your slides and the textbook).

Please, check java file

c) Briefly explain the time and memory complexity of both versions of your calculator. You can write your answer in a separate file and submit it together with the other submissions.

For first version, since we used array-based stack for first version, the memory (space) complexity is O(n). Even if we use 2 stacks, to sum up the space, it becomes the same as input size which is n (how many operators and values exist). Each operation takes O(1) times but as we need to pop out every operators and values, the time complexity is O(n).

Finding more detailed time complexity, except while loop and for loop, all of them are constant time complexity which I denote C_1. For loop runs for size of length of expression (equation), which is input size n, so it's C_2n time complexity. We have two while loop in repeatOps and even in the worst case, one while loop is working and the other one doesn't work. Furthermore, it only calls when it is not a number (when it is operator) so I can denote maximum operator is n/2 so we can denote C_2n time complexity.

To sum them up, it's $C_1+(C_2+C_3)*n \leq c*n$ where $c \geq C_1 + C_2 {}^{+} C_3$ and $n \geq 1$ ($n_0=1$)
Therefore, it is O(n)

On the other hands, second version uses tree and the space complexity is depending on the depth, not the input size. The worst case it can happen is all trees have 2 child 1 + 2 + 4 + … + 2^(log(n))-1 = n (input size) so we know that the depth is log(n) which is the same as memory (space) complexity O(log(n)).

Similarly, to build a tree, it uses while loop and it runs n times. Even though we have recursive call for parenthesis, total calling time does not change. If we denote all other constant times as C_1, then it will be C_1*n times of complexity. For calculating the tree, maximum children can be 2 and each recursive calls 1 + 2 + 4 + … + 2^log(n) = n so can denote this time complexity as C_2*n. Therefore, $C_1n + C_2n \leq c*n$ where $c \geq C_1 + C_2$ and $n \geq 1$ ($n_0=1$) Therefore, it is O(n).

First version: Time complexity O(n), memory complexity O(n)

Second version: Time complexity O(n), memory complexity O(log(n))

d) Provide test logs for the first version for at least 20 different and sufficiently complex arithmetic expressions that use all types of operators (including parentheses) in varying combinations.

Please check java file and run. Console will distribute the result.