You can view this report online at : https://www.hackerrank.com/x/tests/1520410/candidates/49908402/report

Full Name:	Breeha Qasim
Email:	bq08283@st.habib.edu.pk
Test Name:	**CS102 - Lab 4 - Spring 2023**
Taken On:	5 Feb 2023 10:09:16 PKT
Time Taken:	385 min 39 sec/ 5100 min
Work Experience:	< 1 years
Invited by:	Muzammil
Skills Score:	
Tags Score:	

100%
400/400

scored in **CS102 - Lab 4 - Spring 2023** in 385 min 39 sec on 5 Feb 2023 10:09:16 PKT

Recruiter/Team Comments:

No Comments.

Plagiarism flagged

We have marked questions with suspected plagiarism below. Please review.

	Question Description	Time Taken	Score	Status
Q1	Infix to Postfix > Coding	31 min 6 sec	100/ 100	!
Q2	Infix to Prefix > Coding	2 hour 13 min 6 sec	100/ 100	!
Q3	Expression Evaluation > Coding	8 min 40 sec	100/ 100	!
Q4	Prefix Expression Evaluation > Coding	6 min 3 sec	100/ 100	!

QUESTION 1

Needs Review

Score 100

Infix to Postfix > Coding

QUESTION DESCRIPTION

Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions.
In Infix notations, operators are written in-between their operands. However, in Postfix expressions, the operator comes after the operands.

Assume the infix expression is a string of tokens delimited by spaces. The operator tokens are *, /, +, and −, along with the left and right parentheses, (and). The operand tokens are the single-character identifiers A, B, C, and so on.

The following steps will produce a string of tokens in Postfix order.

1. Create an empty stack called op_stack for keeping operators. Create an empty list for output.
2. Convert the input infix string to a list by using the string method split.
3. Scan the token list from left to right.
    - If the token is an operand, append it to the end of the output list.
    - If the token is a left parenthesis, push it on the op_stack.
    - If the token is a right parenthesis, pop the op_stack until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
    - If the token is an operator, *, /, +, or −, push it on the op_stack. However, first remove any operators already on the op_stack that have higher or equal precedence and append them to the output list. When the input expression has been completely processed, check the op_stack. Any operators still on the stack can be removed and appended to the end of the output list.

When the input expression has been completely processed, check the op_stack. Any operators still on the stack can be removed and appended to the end of the output list.

Infix to Postfix Conversion Simulator: https://raj457036.github.io/Simple-Tools/prefixAndPostfixConvertor.html

Write a function `Infix_to_Postfix` that takes an arithmetic expression in Infix notation as a parameter and returns the corresponding arithmetic expression with Postfix notation.

```
>>> Infix_to_Postfix("A + B * C + D")
A B C * + D +

>>> Infix_to_Postfix("( A + B ) * ( C + D )")
A B + C D + *
```

INTERVIEWER GUIDELINES

```
def push(lst, item):
    lst.append(item)


def push(stack,item):
    stack.append(item)


def pop(stack):
    if(len(stack) == 0):
        return None
    return stack.pop()

def top(stack):
    if len(stack) == 0:
        return None
    return stack[-1]

def is_empty(stack):
    if len(stack) == 0:
        return True
    else:
        return False


def has_high_priority(current, topp):
```

```
def has_high_priority(current, topp):
    plist = {'+': 0, '-':0, '*':1, '/':1}

    if plist[current] >= plist[topp]:
        return True
    else:
        return False

def Infix_to_Postfix(expression):

    op_stack=[]
    output =[]



    exp = expression.split()

    for t in exp:

        if t == '(':
            push(op_stack, t)

        elif t == ')':
            while is_empty(op_stack) == False:
                if top(op_stack) != "(":
                    output.append(pop(op_stack))
                else:
                    pop(op_stack)
                    break
        elif t in '+-*/':
            while is_empty(op_stack) == False and top(op_stack) in '+-*/'
and has_high_priority(top(op_stack), t):
                output.append(pop(op_stack))

            push(op_stack, t)
        else:
            output.append(t)

    while is_empty(op_stack) == False:
        output.append(pop(op_stack))

    return ' '.join(output)
```

CANDIDATE ANSWER

Language used: **Python 3**

```
1  def push(lst,item):
2      lst.append(item)
3  def pop(lst):
4      if len(lst)>0:
5          return lst.pop()
6  def top(lst):
7      if len(lst)>0:
8          return lst[-1]
9  def is_empty(lst):
10     if len(lst)>0:
11         return False
12     return True
13
14 def Infix_to_Postfix(expression):
15     op_stack=[] #operator stack
```

```
    operands=[] #operands list
    order_precedence={"*":2,"/":2,"+":1,"-":1,"(":-10,")":-10}
    expression=expression.split()
    for i in expression:
        temp=''
        if i.isalpha()==True:
            operands.append(i)
        elif i=="(":
            push(op_stack,i)
        elif i==")":
            temp=pop(op_stack)
            while temp!="(":
                push(operands,temp)
                temp=pop(op_stack)
        else:
            if is_empty(op_stack)==False:
                while order_precedence[i]<=order_precedence[top(op_stack)]:
                    operands.append(pop(op_stack))
                    if is_empty(op_stack)==True:
                        break
                push(op_stack,i)
            else:
                push(op_stack,i)
    if is_empty(op_stack)==False:
        for i in range(len(op_stack)):
            operands.append(pop(op_stack))
    output=""
    for i in operands:
        output=output+i+" "
    return output
```

TESTCASE	DIFFICULTY	TYPE	STATUS	SCORE	TIME TAKEN	MEMORY USED
Testcase 0	Easy	Sample case	Success	5	0.0954 sec	7.98 KB
Testcase 1	Easy	Sample case	Success	5	0.0311 sec	8.03 KB
Testcase 2	Easy	Sample case	Success	5	0.0281 sec	7.96 KB
Testcase 3	Easy	Sample case	Success	5	0.0292 sec	8.06 KB
Testcase 4	Easy	Sample case	Success	5	0.0319 sec	7.85 KB
Testcase 5	Easy	Sample case	Success	10	0.0657 sec	8.04 KB
Testcase 6	Easy	Sample case	Success	5	0.0613 sec	7.7 KB
Testcase 7	Easy	Sample case	Success	5	0.0389 sec	8.03 KB
Testcase 8	Easy	Sample case	Success	5	0.055 sec	7.98 KB
Testcase 9	Easy	Sample case	Success	5	0.0486 sec	8.02 KB
Testcase 10	Easy	Sample case	Success	5	0.0318 sec	7.88 KB
Testcase 11	Easy	Sample case	Success	5	0.0309 sec	7.8 KB
Testcase 12	Easy	Sample case	Success	5	0.0623 sec	7.71 KB
Testcase 13	Easy	Sample case	Success	10	0.0429 sec	8.03 KB
Testcase 14	Easy	Sample case	Success	5	0.0739 sec	8.01 KB
Testcase 15	Easy	Sample case	Success	5	0.0547 sec	8.07 KB
Testcase 16	Easy	Sample case	Success	5	0.0599 sec	7.75 KB
Testcase 17	Easy	Sample case	Success	5	0.0571 sec	7.91 KB

No Comments

QUESTION 2

Needs Review

Score 100

Infix to Prefix › Coding

QUESTION DESCRIPTION

Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions.
In Infix notations, operators are written in-between their operands. However, in Prefix expressions, the operator comes before the operands.

Assume the infix expression is a string of tokens delimited by spaces. The operator tokens are *, /, +, and −, along with the left and right parentheses, (and). The operand tokens are the single-character identifiers A, B, C, and so on.

The following steps will produce a string of tokens in prefix order:

1. Reverse the infix expression i.e A + B * C will become C * B + A. Note while reversing each '(' will become ')' and each ')' becomes '('.
2. Obtain the postfix expression of the modified expression i.e C B * A +.
3. Reverse the postfix expression. Hence in our example prefix is + A * B C.

Infix to Prefix Conversion Simulator: https://raj457036.github.io/Simple-Tools/prefixAndPostfixConvertor.html

Write a function `Infix_to_Prefix` that takes an arithmetic expression in Infix notation as a parameter and returns the corresponding arithmetic expression with Prefix notation.

```
>>> Infix_to_Prefix("( A + B ) * ( C + D )")
* + A B + C D

>>> Infix_to_Prefix("A * B + C * D")
+ * A B * C D
```

CANDIDATE ANSWER

Language used: **Python 3**

```python
def push(lst,item):
    lst.append(item)
def pop(lst):
    if len(lst) > 0:
        return lst.pop()
def top(lst):
    if len(lst) > 0:
        return lst[-1]

def is_empty(lst):
    if len(lst) == 0:
        return True
    return False


def Infix_to_Prefix(expression):
    precedence = {"+": 1, "-": 1, "*": 2, "/": 2, "(": -10, ")": -10}
    stack = []
    op = []
    expression = expression[::-1]
```

```
    x = expression.split()
    expression = []
    a = []
    b = ""
    for i in x:
        if i == "(":
            expression.append(")")
        elif i == ")":
            expression.append("(")
        else:
            expression.append(i)
    for i in expression:
        z = ""
        if i.isalpha() == True:
            op.append(i)
        elif i == "(":
            push(stack, i)
        elif i == ")":
            z = pop(stack)
            while z != "(":
                push(op, z)
                z = pop(stack)
        else:
            if is_empty(stack) == False:
                while stack != "(" and stack[-1] != ")" and precedence[i] <=
precedence[top(stack)]:
                    op.append(pop(stack))
                    if top(stack) == "(":
                        op.append(pop(stack))
                        break
                    if is_empty(stack) == True:
                        break
                push(stack, i)
            else:
                push(stack, i)
    if is_empty(stack) == False:
        for i in range(len(stack)):
            op.append(pop(stack))

    for i in range(len(op)):
        a.append(pop(op))
    for i in a:
        b += i + " "
    return b
```

TESTCASE	DIFFICULTY	TYPE	STATUS	SCORE	TIME TAKEN	MEMORY USED
Testcase 1	Easy	Sample case	Success	7	0.0546 sec	8.04 KB
Testcase 2	Easy	Sample case	Success	7	0.076 sec	8.07 KB
Testcase 4	Easy	Sample case	Success	7	0.0613 sec	8.09 KB
Testcase 5	Easy	Sample case	Success	7	0.1184 sec	8.1 KB
Testcase 6	Easy	Sample case	Success	7	0.07 sec	8.07 KB
Testcase 7	Easy	Sample case	Success	7	0.1174 sec	8.14 KB
Testcase 8	Easy	Sample case	Success	7	0.0618 sec	8.06 KB
Testcase 9	Easy	Sample case	Success	7	0.0418 sec	8.02 KB
Testcase 10	Easy	Sample case	Success	8	0.0467 sec	7.89 KB
Testcase 11	Easy	Sample case	Success	7	0.0434 sec	8.05 KB
Testcase 12	Easy	Sample case	Success	8	0.0631 sec	8.1 KB

Testcase 13	Easy	Sample case	Success	7	0.1074 sec	8.04 KB
Testcase 14	Easy	Sample case	Success	7	0.0551 sec	8.09 KB
Testcase 15	Easy	Sample case	Success	7	0.0581 sec	8.04 KB

No Comments

QUESTION 3

Needs Review

Score 100

Expression Evaluation › Coding

QUESTION DESCRIPTION

Assume we have a mathematical expression in postfix notation available as a string of operands and operators. The procedure for evaluating such a postfix expression is as follows:

1. Scan the expression left to right.
2. Skip values or variables (operands).
3. When an operator is found, apply the operation to the preceding two operands.
4. Replace the two operands and operator with the calculated value (three symbols are replaced with one operand).
5. Continue scanning until only a value remains -- the result of the expression

Write a program that takes input a well-formed mathematical expression as a string in postfix notation and evaluates it using a stack.
We assume the input to be already in post-fix form.
Also, we are assuming only four binary operators {+, - , /, *} are allowed.

Postfix Expression Evaluation Simulator: https://raj457036.github.io/Simple-Tools/prefixAndPostfixEvaluator.html

```
>>> postfixEval("2 1 - 3 4 2 / + *")
5.0
>>> postfixEval("2 3 4 + * 6 -")
8.0
>>> postfixEval("10.2 8 6 - * 3 / 112.5 +")
119.3
```

CANDIDATE ANSWER

Language used: **Python 3**

```
def push(lst,item):
    lst.append(item)

def pop(lst):
    if len(lst) > 0:
        return lst.pop()
    return None

def top(lst):
    if len(lst) > 0:
        return lst[-1]
    return None

def is_empty(lst):
```

```
    if len(lst) == 0:
        return True
    return False
def enqueue(lst,data):
    lst.appened(data)
def dequeue(lst):
    x=front(lst)
    lst.pop(0)
    return x
def front(lst):
    if len(lst)>0:
        return lst[0]
def postfixEval(exp):
    x=0
    exp=exp.split()
    lst=[]
    operators=["+","*","-","/"]
    for i in range(len(exp)):
        if exp[i] not in operators:
            push(lst,exp[i])
        elif exp[i]=="+":
            x=float(pop(lst))
            y=float(pop(lst))
            z=x+y
            push(lst,z)
        elif exp[i]=="-":
            x=float(pop(lst))
            y=float(pop(lst))
            z=y-x
            push(lst,z)
        elif exp[i]=="*":
            x=float(pop(lst))
            y=float(pop(lst))
            z=y*x
            push(lst,z)
        elif exp[i]=="/":
            x=float(pop(lst))
            y=float(pop(lst))
            z=y/x
            push(lst,z)
    return float(pop(lst))


```

TESTCASE	DIFFICULTY	TYPE	STATUS	SCORE	TIME TAKEN	MEMORY USED
Testcase 0	Easy	Sample case	Success	2	0.0445 sec	8.02 KB
Testcase 1	Easy	Sample case	Success	7	0.0381 sec	8 KB
Testcase 2	Easy	Hidden case	Success	7	0.0667 sec	8.01 KB
Testcase 3	Easy	Hidden case	Success	7	0.0375 sec	7.88 KB
Testcase 4	Easy	Hidden case	Success	7	0.0402 sec	8.06 KB
Testcase 5	Easy	Hidden case	Success	7	0.0384 sec	8 KB
Testcase 6	Easy	Hidden case	Success	7	0.03 sec	7.95 KB
Testcase 7	Easy	Hidden case	Success	7	0.0384 sec	8.1 KB
Testcase 8	Easy	Hidden case	Success	7	0.0403 sec	8.02 KB
Testcase 9	Easy	Sample case	Success	7	0.0465 sec	8.04 KB
Testcase 10	Easy	Hidden case	Success	7	0.0396 sec	7.86 KB
Testcase 11	Easy	Hidden case	Success	7	0.0887 sec	8.02 KB

Testcase 12	Easy	Hidden case	Success	7	0.0329 sec	7.93 KB
Testcase 13	Easy	Hidden case	Success	7	0.0322 sec	8.04 KB
Testcase 14	Easy	Hidden case	Success	7	0.043 sec	7.85 KB

No Comments

QUESTION 4

Needs Review

Score 100

Prefix Expression Evaluation › Coding

QUESTION DESCRIPTION

Assume the prefix expression is a string of tokens delimited by spaces. The operators are *, /, +, and - and the operands are assumed to be single-digit integer values. The output will be an integer result.

1. Create an empty stack called `operandStack`.
2. Convert the string to a list by using the string method `split`.
3. Reverse the prefix expression.
4. Scan the token list from left to right.
   - If the token is an operand, convert it from a string to an integer and push the value onto the `operandStack`.
   - If the token is an operator, i.e. *, /, +, or -, it will need two operands. Pop the `operandStack` twice. The first pop is the first operand and the second pop is the second operand. Perform the arithmetic operation. Push the result back on the `operandStack`.
5. When the input expression has been completely processed, the result is on the stack. Pop the `operandStack` and return the value.

Prefix Expression Evaluation Simulator: https://www.free-online-calculator-use.com/prefix-evaluator.html

Write a function `EvaluatePrefix` that take, as input, an expression in prefix notation, and return, as output, the computed value of the given expression.

INTERVIEWER GUIDELINES

```
def push(lst, item):
    lst.append(item)

def top(lst):
    if is_empty(lst) == False:
        return lst[-1]

def pop(lst):
    val = lst[-1]
    lst.pop(-1)
    return val

def is_empty(lst):
    if lst == []:
        return True
    else:
        return False

def string_reversal(s):
    lst = []
    for letter in s:
        push(lst,letter)
    reverse_string = ""
    while is_empty(lst) == False:
        letter = pop(lst)
```

```
        letter = pop(lst)
        reverse_string = reverse_string + letter
    return reverse_string


num_lst = ["1","2","3","4","5","6","7","8","9","0"]

def EvalutePrefix(expression):
    operandStack = []
    reverse_string = string_reversal(expression)
    str_lst = reverse_string.split()

    for item in str_lst:
        if item in num_lst:
            item = int(item)
            push(operandStack,item)
        elif item == "+":
            num1 = pop(operandStack)
            num2 = pop(operandStack)
            result = num1 + num2
            push(operandStack,result)
        elif item == "-":
            num1 = pop(operandStack)
            num2 = pop(operandStack)
            result = num1 - num2
            push(operandStack,result)
        elif item == "/":
            num1 = pop(operandStack)
            num2 = pop(operandStack)
            result = int(num1 / num2)
            push(operandStack,result)
        elif item == "*":
            num1 = pop(operandStack)
            num2 = pop(operandStack)
            result = num1 * num2
            push(operandStack,result)
    return top(operandStack)
```

CANDIDATE ANSWER

Language used: **Python 3**

```
def push(lst,item):
    lst.append(item)

def pop(lst):
    if len(lst) > 0:
        return lst.pop()
    return None

def top(lst):
    if len(lst) > 0:
        return lst[-1]
    return None

def is_empty(lst):
    if len(lst) == 0:
        return True
    return False

def EvalutePrefix(expression):
    operandStack=[]
    expression=expression.split()
```

```
    expression=expression[::-1]
    operator=["+","*","-","/"]
    for i in range(len(expression)):
        if expression[i] not in operator:
            push(operandStack,float(expression[i]))
        elif expression[i]=="+":
            x=float(pop(operandStack))
            y=float(pop(operandStack))
            z=x+y
            push(operandStack,z)
        elif expression[i]=="-":
            x=float(pop(operandStack))
            y=float(pop(operandStack))
            z=abs(y-x)
            push(operandStack,z)
        elif expression[i]=="*":
            x=float(pop(operandStack))
            y=float(pop(operandStack))
            z=x*y
            push(operandStack,z)
        elif expression[i]=="/":
            x=float(pop(operandStack))
            y=float(pop(operandStack))
            z=x/y
            push(operandStack,z)
    return int(pop(operandStack))
```

TESTCASE	DIFFICULTY	TYPE	STATUS	SCORE	TIME TAKEN	MEMORY USED
Testcase 0	Easy	Sample case	Success	6	0.094 sec	8.02 KB
Testcase 1	Easy	Sample case	Success	6	0.0368 sec	7.96 KB
Testcase 2	Easy	Sample case	Success	8	0.0353 sec	7.95 KB
Testcase 3	Easy	Sample case	Success	8	0.041 sec	7.96 KB
Testcase 4	Easy	Sample case	Success	8	0.0422 sec	8.02 KB
Testcase 5	Easy	Sample case	Success	8	0.0826 sec	7.92 KB
Testcase 6	Easy	Sample case	Success	8	0.0789 sec	7.89 KB
Testcase 7	Easy	Sample case	Success	8	0.0317 sec	7.82 KB
Testcase 8	Easy	Sample case	Success	8	0.0385 sec	7.91 KB
Testcase 9	Easy	Sample case	Success	8	0.0309 sec	7.95 KB
Testcase 10	Easy	Sample case	Success	8	0.0484 sec	7.96 KB
Testcase 11	Easy	Sample case	Success	8	0.0332 sec	7.9 KB
Testcase 12	Easy	Sample case	Success	8	0.0418 sec	8.01 KB

No Comments