CS211 - Data Structures and Algorithms

Lab 08

Instructor: Dr. Sharaf Hussain
E-mail: shhussain@uit.edu

Semester: Fall, 2021

1 Objective

The purpose of this lab session is to implement **Stacks** data structure and its applications.

2 Instructions

You have to perform the following tasks yourselves. Raise your hand if you face any difficulty in understanding and solving these tasks. **Plagiarism** is an abhorrent practice and you should not engage in it.

3 How to Submit

- Submit lab work in a single .py file on Google Classroom. (No other format will be accepted)
- Lab work file name should be saved with your roll number (e.g. 19a-001-SE_LW04.py)
- Submit home work in a single .py file on Google Classroom. (No other format will be accepted)
- Lab work file name should be saved with your roll number (e.g. 19a-001-SE_HW04.py)

Task 1 - Stack

Implement following **Stack** using linked list data structure.

```
#stack ADT using linked list
class Stack:
  def __init__(self):
      self._top = None
      self._size = 0

  def isEmpty(self):
      return self._top is None

  def __len__(self):
      return self._size

  def peek(self):
      assert not self.isEmpty(), "Empty stack"
      return self._top.item

  def pop(self):
      assert not self.isEmpty(), "Empty stack"
      node = self._top
      self._top = self._top.next
      self._size -= 1
      return node.item

  def push(self, item):
```

```
        self._top = _StackNode(item, self._top)
        self._size += 1

    class _StackNode:
    def __init__(self, item, link):
        self.item = item
        self.next = link
```

Task 2 - Infix to Postfix

Implement following **Infix to Postfix** algorithm using Stack data structure.

```
from Chapter7.lliststack import Stack

def infixToPostfix(infixexpr):
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    opStack = Stack()
    postfixList = []
    tokenList = infixexpr.split()
    print("Token List: ",tokenList)
    for token in tokenList:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":
            postfixList.append(token)
            print('postfixList: ',postfixList)
        elif token == '(':
            opStack.push(token)
            print('Taken is (, push into opStack: ', opStack.peek())

        elif token == ')':
            print('Taken is ), pop opStack: ')
            topToken = opStack.pop()

            while topToken != '(':
                postfixList.append(topToken)
                topToken = opStack.pop()
        else:
            while (not opStack.isEmpty()) and (prec[opStack.peek()] >= prec[
    token]):
                print('opStack peek %c >= token %c, put %c in the
    postfixList : ' %(opStack.peek(),token,opStack.peek()))
                postfixList.append(opStack.pop())
            opStack.push(token)
            print('token is %c, push into opStack: '%(token))

    while not opStack.isEmpty():
        print('Stack Peek put into postfixList: ', opStack.peek())
        postfixList.append(opStack.pop())

    return " ".join(postfixList)
```

Task 3 - Evaluate Postfix

Implement following **Postfix Evaluation** algorithm using Stack data structure.

```
from Chapter7.lliststack import Stack
def evaluatePostfix(e):
    #  Get the length of expression
    size = len(e)
    a = 0
    b = 0
    s = Stack()
```

```
    isVaild = True
    i = 0
    #  work with (+,-,/,*,%) operator
    while (i < size and isVaild):
        if (e[i] >= '0'
                and e[i] <= '9'):
            a = ord(e[i]) - ord('0')
            s.push(a)
        elif (len(s) > 1):
            a = s.pop()
            b = s.pop()
            #  Perform arithmetic operations between 2 operands
            if (e[i] == '+'):
                s.push(b + a)
            elif (e[i] == '-'):
                s.push(b - a)
            elif (e[i] == '*'):
                s.push(b * a)
            elif (e[i] == '/'):
                s.push(int(b / a))
            elif (e[i] == '%'):
                s.push(b % a)
            else:
                #  When use other operator
                isVaild = False

        elif (len(s) == 1):
            #  Special case
            #  When use +, - at the beginning
            if (e[i] == '-'):
                a = s.pop()
                s.push(-a)
            elif (e[i] != '+'):
                #  When not use  +,-
                isVaild = False

        else:
            isVaild = False

        i += 1

    if (isVaild == False):
        #  Possible case use other operators
        #  1) When using special operators
        #  2) Or expression is invalid
        print(e, " Invalid expression ")
        return

    print(e, " = ", s.pop())
```

4 Student's Tasks

Task 1

Write and test a program that extracts postfix expressions from the user, evaluates the expression, and prints the results. You may require that the user enter numeric values along with the operators and that each component of the expression be separated with white space.

Task 2

We can design and build a postfix calculator that can be used to perform simple arithmetic operations. The calculator consists of a single storage component that consists of an operand stack. The operations performed by the stack always use the top two values of the stack and store the result back on the top of the stack. Implement the operations of the Postfix Calculator ADT as de
ned here:.

- `PostfixCalculator()`: Creates a new post

x calculator with an empty operand stack.

- `value( x )`: Pushes the given operand x onto the top of the stack.
- `result()`: Returns an alias to the value currently on top of the stack. If the stack is empty, `None` is returned.
- `clear()`: Clears the entire contents of the stack.
- `clearLast()`: Removes the top entry from the stack and discards it.
- `compute( op )`: Removes the top two values from the stack and applies the given operation on those values. The first value removed from the stack is the righthand side operand and the second is the lefthand side operand. The result of the operation is pushed back onto the stack. The operation is specified as a string containing one of the operators + - * / **.
- **Extend the Postfix Calculator ADT as follows:**
  (a) To perform several unary operations commonly found on scientific calculators: absolute value, square root, sine, cosine, and tangent. The operations should be specified to the `compute()` method using the following acronyms: `abs, sqrt, sin, cos, tan`.
  (b) To use a second stack on which values can be saved. Add the following two operations to the ADT:
  - `store()`: Removes the top value from the operand stack and pushes it onto the save stack.
  - `recall()`: Removes the top value from the save stack and pushes it onto the operand stack.

Task 3

Implement a complete maze solving application using the components introduced earlier in the chapter. Modify the solve() method to return a vector containing tuples representing the path through the maze.