FIT1008– Intro to Computer Science
Workshop Week 10
Semester 1, 2017

Objectives of this practical session

The aim of this prac is to gain experience in implementing Linked Structures. The first part is about comparing Array and Linked Implementations of Stacks and Queues. The second part implements a reverse polish notation calculator. [1]

[1] We have introduced a checkpoint in this workshop to serve as a guideline for the amount of work you should do before the lab session. Aim for it or beyond to make the most out of your prac.

Task 1

- Implement an array based implementation of the Stack ADT
- Implement an linked implementation of the Stack ADT
- Using the methodology studied in the Workshop of week 5, time the running time of the main ADT operations and compare how the two implementations behave for large input sizes. How can you relate your findings to the time complexity of the operations?

Task 2

- Implement an array based implementation of the Queue ADT.
- Implement an linked implementation of the Queue ADT
- Using the methodology studied in the Workshop of week 5, time the running time of the main ADT operations and compare how the two implementations behave for large input sizes. How can you relate your findings to the time complexity of the operations?

Reverse Polish Notation

Reverse Polish Notation (RPN) was invented in the 1920's by Polish mathematician Jan Lukasiewicz as a different way of entering mathematical expressions by writing the operator after the operands instead of between them. It does not have the precedence problems that "infix" notation does and can therefore avoid the use of parenthesis.

The following table illustrates how "infix" expressions can be written in Reverse Polish notation:

Infix	Reverse Polish
a + b*c	a b c * +
a + (b*c)	a b c * +
(a + b)*c	a b + c *
a*b + c	a b * c +
a*(b + c)	a b c + *
(a + b)*(c - d)	a b + c d - *
(a + b)*(c - d)/(e + f)	a b + c d - * e f + /

Note that

- the variables appear in the same order
- the operators (+, *, -, /) come after the operands (variables) they are operating on
- no parentheses are needed

In this rest of this prac we will use a stack to evaluate the Reverse Polish integer expressions. The idea is as follows: as each integer is read from the screen, it is pushed onto the stack; when the characters +, -, * or / are read, two integers are popped off the stack, the operator is applied to them, and the result is pushed back onto the stack; when the end of the string is read this marks the end of the expression to be evaluated and the integer on top of stack is popped out and printed; if the stack contains anything else at this point, an error message is printed.

For example, the infix expression

$$(1 - 7) * (4 + 5)$$

should be entered via the screen as

```
1 7 - 4 5 + *
```

When 1 is read it is pushed onto the stack. Then 7 is read and also pushed onto the stack. Then, the character - is read, 7 and 1 are popped off, 7 is subtracted from 1, and the result -6 is pushed onto the stack. Next 4 is read and pushed onto the stack, and 5 is read and also pushed onto the stack. Then, the character + is read, 5 and 4 are popped off and their sum 9 is pushed onto the stack. Then, after the character * is read, 9 and -6 are popped off the stack, and their product -54 is pushed onto the stack. Finally, once the end of the string is read, -54 is popped off the stack and printed. Since the stack is empty, no error is given.

Task 3

Before you can write a program to evaluate expressions you need to be able to interpret the input.

Write a Python program that prompts the user for a character string, splits it into its substrings (using spaces as the delimiters), and returns each substring together with a message indicating whether the substring is an integer, an operator (accepted operators will be addition, subtraction, multiplication, division, and equality, i.e., '+', '-', '*', '/'), or an invalid string.

For example, for the following input string:

```
12 3 + -8ha -98
```

the output would be:

```
12 Integer
 3 Integer
```

```
  + Operator
-8ha Invalid string
 -98 Integer
```

Tip: You should divide your program into separate chunks of code. For instance, you could delegate the part of the code that takes a string and generates a description to its own separate `describe()` function. This also allows you to run tests on sub-functionality of your program.

CHECKPOINT

(You should aim to reach at least this task **before** the lab session)

Task 4

Write a Python program that allows a user provide an integer expressions in Reverse Polish Notation. For each expression the user provides the program evaluates the expression and returns the result.

Important: Make sure your program deals with invalid input, i.e., detects invalid input, gives the appropriate message and allows the program to continue. Invalid inputs includes strings that are too large for the stack, and those containing too many operands/operators.

Optional – Task 5

So far our simple calculator reads expressions in Reverse Polish notation. In this question you will write a function that allows you to read integer infix expressions and prints out Reverse Polish expressions.

Dijkstra's shunting algorithm is a method of generating Reverse Polish notation from normal "infix" notation. We say that infix multiplication and division have a higher precedence than addition and subtraction. The former bind more tightly to their operands than the latter. The shunting algorithm uses a stack of characters and works as follows. When an INTEGER is read in, it is immediately printed. When an OPERATOR is read in, (let's call it the current Operator), then either:

- the stack is empty and current Operator is pushed onto the stack,
- the current Operator has higher precedence than the OPERATOR on top of the stack, and the current Operator is pushed onto the stack, or
- the current Operator has lower, or equal, precedence than the OPERATOR on top of the stack, and in this case those OPERATORS on the stack with greater, or equal, precedence than the current Operator are popped off the stack (and printed) until either the stack is empty, or the OPERATOR on top of the stack has lower precedence than the current Operator. Then the current Operator is pushed onto the stack.

Finally, when all the input is exhausted all the OPERATORS remaining on the stack are popped off and printed. For example, consider the following infix expression: 4*3+6/2 , which is equivalent to (4*3)+(6/2), evaluates to 15, and is written in Reversed Polish Notation as 4 3 * 6 2 / +. The shunting algorithm:

- reads a 4 and immediately prints it,
- reads a * and pushes it into the stack (since it is empty),
- reads a 3 and immediately prints it,
- reads a + and since + has lower precedence than the top of stack *, it pops *, prints it, and then pushes +,

- reads a 6 and prints it,
- reads a / and since it has higher precedence than the top of stack +, it pushes / into the stack,
- reads a 2 and immediately prints it
- since the input is exhausted, it pops / and prints it, and then pops + and prints it

Write a Python program that reads integer infix expressions and prints out Reverse Polish expressions to the screen.