Data Structures: Problem Set 1

Serge Salan

due September 2 at 11:59pm

1 Evaluating arithmetic expressions

Java classes: ArrayStack, Arithmetic, Test.

The objective is to show how the stack data structure can be used to efficiently evaluate arithmetic expressions. A class ArrayStack is provided and should not be modified. A class Test contains test cases with their expected output. Write your solution in the class Arithmetic. Your solution must run in linear time.

The input to the program is a string that consists of an *arithmetic expression*. The string can only contain operators $+ - */$, parentheses (), and operands which are integer and fractional quantities, e.g $200, -12, 44.7, -62.19$. We assume that every operation is contained inside parentheses, with the exception of the outer most operation. We also separate the elements of the expression with spaces. Some input examples:

```
String exp1 = "-4.1 + 8";
String exp2 = "( 2.2 - 7 ) / 5";
String exp3 = "( 27 / 3 ) + ( 2 * -4 )";
String exp4 = "4 / ( ( 3 * ( 5.5 * 2 ) ) + ( ( 27 / 3 ) + ( 2 * -4 ) ) )";
```

1. Write a function *split* that takes an arithmetic expression and places each element of the expression in an array of strings. For example, for the input $exp2$ we obtain the array $[(, 2.2, -, 7,), /, 5]$. In your solution you are asked not to use the built-in Java function *String.split*.

   [3 points]

2. The expression $(2.2 - 7)/5$ is written in *infix notation*, where operators are placed between operands. There exists alternative notations to write arithmetic expressions. In the *postfix notation*, operators are placed after their operands, e.g. $2.2\ 7 - 5/$. Parentheses are not needed in the postfix notation.

   Write a function $infixToPostfix$ that uses a stack to convert an expression in infix notation to postfix. The input and the output are arrays of strings. For the same example, we obtain the array $[2.2, 7, -, 5, /]$.

   [3 points]

3. Write a function $evaluatePostfix$ that uses a stack to find the value of an expression written in postfix notation. The input is an array of strings For the same example, we obtain the value -0.96.

   [3 points]

4. Write a function *evaluate* that combines the steps from questions 1-3 in one function. The function uses two stacks, but does not create any additional arrays. The input is a string that consists of an arithmetic expression. For example the input $exp2$ returns the value -0.96.

   [3 points]

2 Array queues

Java classes: ArrayQueue, ReversibleArrayQueue, Test2.

The objective is to provide more functionality to the array queue data structure. Test2 contains test cases with their expected output. Write your solution in the classes ArrayQueue and ReversibleArrayQueue.

1. In ArrayQueue, write a function *copy* that creates a copy of the array queue and returns it. The copy is a new object of ArrayQueue and contains the same values found in the original queue. The values must be added in the correct order, i.e. starting at the head and ending at the tail.

   [3 points]

2. In ArrayQueue write a function *enhancedEnqueue* that operates exactly like *enqueue* with the following improvement. If the queue is full, the array length is doubled to allow adding new values. This can be achieved by creating a new array, copying the values to the new array, and reassiging the three instance variables *arr*, *head*, and *tail*.

   [3 points]

3. Write a class *ReversibleArrayQueue* that is similar to *ArrayQueue* but contains an additional function *reverse*. When *reverse* is called, the order of the values in the queue is inverted. The following example shows operations made on a reversible array queue.

```
enqueue 5          [ 5 ]
enqueue 4          [ 5 4 ]
reverse            [ 4 5 ]
enqueue 8          [ 4 5 8 ]
enqueue 2          [ 4 5 8 2 ]
dequeue --> 4      [ 5 8 2 ]
enqueue 7          [ 5 8 2 7 ]
reverse            [ 7 2 8 5 ]
dequeue --> 7      [ 2 8 5 ]
dequeue --> 2      [ 8 5 ]
dequeue --> 8      [ 5 ]
dequeue --> 5      [ ]
```

   The class must have the functions *enqueue*, *dequeue*, and *reverse* implemented in constant time. Add new instance variables if needed.

   [6 points]