Stacks and Queues

Updated: 7th December, 2021

Aims

- To familiarise yourself with UML diagrams.
- To create general purpose Stack and Queue classes, with test harnesses to test their functionality.
- Apply these classes to imitate and understand the Call Stack.
- To implement a program to convert Infix equations to Postfix and evaluate the Postfix equation.

Before the Practical

- Read this practical sheet fully before starting.
- Download and read the *"UML Style Guide"* located under *Links and Resources on Blackboard*.
- Download and read *"Test Harness Information from Intro to SE"* located under *Links and Resources on Blackboard*.

Activities

1. UML Diagrams

Before you start coding, draw the UML diagrams for `DSAStack`, `DSAQueue`, `DSAShufflingQueue`, `DSACircularQueue` and their test harnesses. These can be done by hand or via an application such as *Microsoft Visio* or *diagrams.net* (formerly known as *draw.io* and saved as a .pdf or .png).

Get feedback on your diagrams before you start coding.

2. Implementation of Stack and Queue

Although Java and Python provide Stack and Queue classes to implement these abstract data types, we will be implementing our own versions to get a hands-on understanding of how they work.

- Create two classes: `DSAStack` and `DSAQueue` and implement them using **arrays** as the data structure.
- Use your UML diagrams and the pseudocode from the lecture notes to guide you in their implementation.
- Write test harnesses to fully test both classes, ensuring exception handling is implemented.

Note:

- Use an *Object[ ] / dtype=object* array rather than the *double* from the lecture notes.

  Object is a special class in Java/Python in that *all* classes inherit from Object. (*i.e.,* All classes are a specialisation of Object)

  Making the array type **Object[]** means that we don't limit what kind of data we can put into the ADT, thus creating a *general-purpose* ADT class.
- Use an *int* member field to track the count of elements in your classes. A value of *0* implies that your classes are empty.
- Using an array to store elements in the `DSAQueue` is more problematic than it is for `DSAStack` because `DSAQueue's` front and rear both *"move"*.

  When enqueuing a new element to the rear of the DSAQueue, the current tail index increases by one. Conversely, when dequeuing the element at the front of the DSAQueue, the current head index also changes.
  - The first solution we will use is to shift all the elements up by one after dequeuing the first value, causing a *shuffling* effect. This *Shuffling Queue* is simple, but inefficient
- Java Students: Do not implement copy constructors for DSAStack and DSAQueue - it is usually both unnecessary and inappropriate to make copies of lists.

3. Circular Queue

- Create a second implementation of the `DSAQueue` to use a *Circular Queue*.
- Modify your test harness to fully test both versions of the queues.

Note:

- The *Circular Queue* "chases the front" and cycles around the array when the front passes the "end" of the array. Be sure to track the count of elements in the queue, although you /emphcould calculate the count from the "start" and "end" indexes, it makes the queue /emphmuch more to get working right.
- Set the *Circular Queue* and *Shuffling Queue* up as sub-classes of the `DSAQueue` parent class. See the lecture slides for the simplified UML.

4. Call Stack

Using your `DSAStack`, you will write some code to simulate the *call stack* as introduced during the previous practical on *Recursion*.

- Your code should call your previously implemented iterative/recursive methods and store the method names and arguments onto the stack as a *string*. As you exit the method, these values are popped off the stack.
- Your code should be able to display the *call stack* at anytime by calling a `display()` method.
- Create a test harness with iterative calls and recursive calls, ensuring exception handling is implemented.

5. Equation Solver

You are going to write a program that can take a *string* representing a mathematical equation in *infix* form and solve it by converting it into *postfix* and then evaluating the *postfix*.

The `EquationSolver.java/.py` file should end up having at least the following methods:

```
public double solve(String equation)
OR
def solve(equation):

Should call parseInfixToPostfix() then evaluatePostfix().
```

```
private DSAQueue parseInfixToPostfix(String equation)
OR
def _parseInfixToPostfix(equation):

Converts infix form equation into postfix.
Stores the postfix terms into a queue as Objects.
Use Doubles for operands and Characters for operators, but put them all into
the queue in postfix order.
(See below for hints on parsing infix to postfix.)
```

```
private double evaluatePostfix(DSAQueue postfixQueue)
OR
def _evaluatePostfix(postfixQueue):

Takes the postfixQueue and evaluates it.
[Java] Use instanceof to determine if a term is an operand (Double) or an
operator (Character).
(See below for hints on evaluating postfix)
```

```
private int precedenceOf(char theOp)
OR
def _precedenceOf(theOp):

Helper function for parseInfixToPostfix().
Returns the precendence (as an integer) of the Operator
i.e.,
+, -
        return 1
*, /
        return 2
```

```
private double executeOperation(char op, double op1, double op2)
OR
def _executeOperation(op, op1, op2):

Helper function for evaluatePostfix().
Executes the binary operation implied by op:
op1 + op2
op1 - op2
op1 * op2
op1 / op2
        return result
```

Note:
parseInfixToPostfix()

- See the lecture notes for pseudocode - however, note that the pseudocode will need some thinking to get it working correctly in real code!
- Assume that the passed-in equation has spaces between **all** operators, operands and brackets - that'll make life much easier.
  Use split, giving delim as `" "` - a space.
- Use a `DSAStack` to stack up the operators.
  Note: It is not possible to store a *char* because it is a primitive, not an *Object*
- When you get a token, look at the first character and perform a switch statement on this.
  - If the token is an operator (+-*/), pop off all operators on the stack that are equal or higher precedence (use `precendenceOf()` and `DSAStack.top()` to check) and enqueue them to the postfixQueue.

    Then push the new operator onto the stack. You will need to pop off ones of equal precedence since they should be done in left-to-right order (only important for '-' and '/' since *a - b != b - a*, so if you allowed the order to reverse, it will stuff you up.)

Note:

- – Make sure you don't pop off '(' since that's the start of the current sub-equation, and only gets popped off when the corresponding ')' is found.
  - – If the token is a '(', push it onto the stack with no other processing.
  - – If the token is a ')', pop operators off the stack and *enqueue* them onto the postfixQueue until the corresponding '(' is found.
    Pop the ')', but don't enqueue it.
  - – Otherwise the token must be a number, so use `Double.valueOf()` on the full token string to convert it into a Double and enqueue it onto the postfixQueue.
- When there are no more tokens, transfer the remaining operators on the stack to the postfixQueue in pop() - order.
- Non-delimiter tokens are numbers. Work with **Doubles** rather than Ints so that you can handle decimals.
  Use `[Java] Double.valueOf()` or `[Python] float()` to convert the string version of a number into a true double.
- You may assume that negative numbers aren't allowed - this avoids confusion between the binary operation *minus* and the unary operation *negate*.
- Don't worry about checking the syntax of the equation - just throw exceptions if something goes wrong, such as no associated '(' for a ')', or non-numeric terms.

Before you get onto `evaluatePostfix()`, make sure that your infix-to-postfix is working by printing out the contents of the postfixQueue returned by `parseInfixToPostfix()` and running it against a couple of examples.

Note:
evaluatePostfix()

- You will need a `DSAStack` to hold the *operands* for evaluating (in parsing, it was the *operators* that were stacked, but here it is the *operands*
- If an item is `[Java] instanceof Character`, it is an operator - grab the top two operands from the stack and evaluate the binary operation.
  Push the result back onto the operandStack.
- Note: The first operand from the stack is also the first operand in the binary operation. (i.e., To the *left* of the operation.)
  For + and *, you won't see any difference but for - and /, getting the order reversed will make a big difference!

After the postfixQueue has been finished, there should only be one operand left on the operandStack - this will be the final solution.

Submission Deliverable

- Your code is due 2 weeks from your current tutorial session.
  - You will demonstrate your work to your tutors during that session
  - If you have completed the practical earlier, you can demonstrate your work during the next session
- You must **submit** your code and any test data that you have been using **electronically via Blackboard** under the *Assessments* section before your demonstration.
  - Java students, please do not submit the *.class files

Marking Guide

Your submission will be marked as follows:

- [2] Your UML diagrams for your stack, queues and test harnesses.
- [2] Your DSAStack is implemented properly - show this through your test harness.
- [2] Your DSAQueue is implemented properly as both Shuffling and Circular with polymorphism - show this through your test harnesses.
- [2] Your Call Stack program is implemented correctly - show this through your test harness.
- [2] The Equation Solver has been implemented properly and uses your stack and queue.

End of Worksheet