CURTIN UNIVERSITY (CRICOS number: 00301J)

Department of Computing, Faculty of Engineering and Science

Data Structures and Algorithms (DSA1002)

PRACTICAL 3 - STACKS & QUEUES

AIMS

- To create general-purpose stack and queue classes
- 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.

ACTIVITY 1: UML DIAGRAMS

Before you start coding, draw the UML diagrams for DSAStack, DSAQueue, and DSAShufflingQueue, DSACircularQueue their test harnesses. Get feedback on the diagrams before you start coding.

ACTIVITY 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.
- Use an *Object[ ] / dtype=object* array rather than the double of 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). Thus making the array type **Object**[ ] means that you don't limit what kind of data you can put into the ADT – we are creating *general-purpose* ADT classes here.
- Use an int member field to track the count of elements in the stack. Thus a value of 0 implies that the stack is empty.
- Note that using an array to store the elements in the DSAQueue is more problematic than it is for DSAStack because DSAQueue's front and rear both 'move'. That is, when enqueuing a new element to the rear the current tail index increases by one, and when dequeuing the element at the front 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 queu 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.

Make sure you write test harnesses and test both classes fully.

ACTIVITY 3: CIRCULAR QUEUE

- Create a second implementation of the DSAQueue to use a circular queue.
- The circular queue 'chases the front' and even cycles around the array when the front passes the 'end' of the array. Just be sure to track the count of elements in the queue: although you *could* calculate the count from the 'start' and 'end' indexes, it makes the queue *much* more complicated to get working right.
- Set this and the shuffling queue up as sub-classes of the DSAQueue parent class. See the lecture slides for the simplified UML.
- Modify your test harness to test out both versions of the queues.

ACTIVITY 4: CALL STACK

- We are now going to write some code to imitate the call stack. In your code, when you enter a method, its name and arguments are put onto the stack as a string. As you exit the method, those values are popped back off the stack. You can display the stack at any time by calling its display method.
- This is similar to a question on recent mid-semester tests.
- Provide test cases with nested method calls and with a recursive call.

ACTIVITY 5: EQUATION SOLVER

You are going to write a simple class that can take a string representing a maths equation in infix form and solve it (calculate it) by converting it to postfix and then evaluating the postfix.

- Create an EquationSolver.java/.py file. The class will end up having at least the following methods (no class fields are needed – we don't need to store anything)

  - `public double solve(String equation)` OR `def solve(equation):`
    Should call parseInfixToPostfix() then evaluatePostfix()

  - `private DSAQueue parseInfixToPostfix(String equation)`
    OR `def _parseInfixToPostfix(equation)`
    Convert infix-form equation into postfix. Store the postfix terms in a queue of Objects; use Doubles for operands and Characters for operators, but put them all in the queue in postfix order. See below for hints on parsing infix to postfix.

  - `private double evaluatePostfix(DSAQueue postfixQueue)`
    OR `def _evaluatePostfix(postfixQueue)`
    Take the postfixQueue and evaluate it. [Java] Use **instanceof** to determine if a term is an operand (Double) or an operator (Character).

- private int precedenceOf(char theOp) OR def _precedenceOf(theOp)
  Helper function for parseInfixToPostfix(). Returns the precedence (as an integer) of theOperator (ie: +,- return 1 and *,/ 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, either op1 + op2, op1 – op2, op1 * op2 or op1 / op2, and returns the result.

parseInfixToPostfix() hints:

- 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 that 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 of *equal or higher* precedence (use precedenceOf() and DSAStack.top() to check) and enqueue them to the postfixQueue. Then push the new operator onto the stack. You 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)
  - Make sure you don't pop off '(' since that's the start of the current subequation, and only gets popped 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 of 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 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 Double.valueOf() [Java] or float() [Python] to convert the string version of a number into a true double.
- You may assume 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 on to evaluatePostfix(), just 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.

evaluatePostfix() hints:

- You will need an DSAStack to hold the *operands* for evaluating (in parsing, it was the *operators* that were stacked, but here it is the *operands*).
- Process each item on the passed-in postfixQueue in FIFO order.
  - If an item is instanceof Double [Java] , it is an operand – push it onto the operandStack
  - If an item is instanceof Character [Java], it is an operator – grab the top two operands from the stack and evaluate the binary operation. Push the result back on to the operandStack.
  - Note that 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.
Create a test harness for the solver.

MARKING GUIDE

Your submission will be marked as follows:

- [2] You have 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 harness.
- [2] Your Call Stack program is implemented correctly.
- [2] The Equation Solver has been implemented properly and uses your stack and queue.

SUBMISSION DELIVERABLE:

Your classes (all that are required for this program) are due before the beginning of your next tutorial. Also include any other relevant classes that you may have created.
SUBMIT ELECTRONICALLY VIA MOODLE, under the *Assessments* section.