

CURTIN UNIVERSITY (CRICOS number: 00301J)
Faculty of Engineering and Science
Department of Computing
Data Structures and Algorithms

Practical 4

Aims

- To create general-purpose stack and queue classes for later use in the DSAMining application.
- 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:

If you look closely at ShedIron and ShedNickel, you will see that they need to have special data structures to handle the shed's OrePile contents. In particular, ShedIron organises its piles into a **queue** and ShedNickel uses a **stack**. In order to implement these classes, we first have to implement data structures for manipulating stacks and queues. These do not appear in the UML since they are very standard classes rather than application-specific, and the UML designer wants to give the programmer leeway as to exactly how to implement the lists.

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

- Create two classes: DSASStack and DSAQueue and implement them in Java using arrays as the data structure. Use the pseudocode from the lecture notes to guide you in their implementation (try to avoid the book's pseudocode on this because it uses tricks that won't make much sense to you at this point in your career!).
- Use an `Object[]` array rather than the `double` of the lecture notes. `Object` is a class in Java, just like `String`. However it is special in that Java forces *all* classes to inherit from `Object` (ie: 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 which index represents 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 DSASStack because DSAQueue's front and rear both 'move'. That is, when enqueueing 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 solution that the lecture notes take is to shift all the elements up by one after dequeuing the first value, causing a *shuffling* effect. This is simple but is admittedly a little inefficient.

- A more efficient alternative is a circular queue, which '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.
- It is suggested that you implement the shuffling queue since your understanding of a queue's workings is more important than the efficiency of its implementation – *you will be tested on shuffling queues*, so make sure you understand them!

Do not implement copy constructors for DSAShuffle and DSAQueue – it is usually both unnecessary and inappropriate to make copies of lists.

Make sure you write a test harness and test both classes fully.

Activity 2: Create EquationSolver class

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 file. The class will end up having at least the following methods (no classfields are needed – we don't need to store anything long term)
 - `public double solve(String equation);`
Should call `parseInfixToPostfix()` then `evaluatePostfix()`
 - `private DSAQueue parseInfixToPostfix(String 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)`
Take the postfixQueue and evaluate it. Use **instanceof** to determine if a term is an operand (Double) or an operator (Character). See below for hints on evaluating it.
 - `private int precedenceOf(char theOperator);`
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);`
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 (Java) code!
- Assume that the passed-in equation has no spaces – that'll make life much easier!
- Use StringTokenizer, giving delims as "+-*/()", **and return the delims too** (since they are also important). You'll need to import `java.util.*`; for StringTokenizer. In other words:

```
StringTokenizer strTok = new StringTokenizer(equation, "+-*/()", true);
```
- Use a DSAShuffle 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 (using `.charAt(0)` on the token string) 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 `DSASStack.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 $a - b \neq b - a$, 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()` to convert the string version of a number into a true double. (`Double.parseDouble()` does a similar job, but returns a double rather than a Double, and so can't be added as an Object to the postfix DSAQueue).
- 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 `DSASStack` 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`, it is an operand – push it onto the operandStack
 - If an item is instanceof `Character`, 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 (ie: 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.

Activity 4: System Testing

Create a `main()` that asks the user to input an equation, use `EquationSolver.solve()` to evaluate it and print out the result. Don't put spaces in the equation unless you specifically handled them in your `EquationSolver` code!

Test it out with various equations. Start small, say $3 * 4$, and build up to large equations with nested brackets, etc.

Submission Deliverable:

You have until your next week's practical to get `DSASStack`, `DSAQueue` and `EquationSolver` implemented and working, with the code commented and cleaned up to your satisfaction. You must then upload your Java source code into the submission area at the *beginning* of the prac session.

Only the source code is required, but you may include your pseudocode as well, and/or a `readme.txt` that provides any important information you feel the assessor should know about.

SUBMIT ELECTRONICALLY VIA BLACKBOARD, under the *Assessments* section.

If you finish early, use the rest of the practical to start the next worksheet, because that will be due later on.