Motivation

Have you ever thought about how you actually do arithmetic? For example, how exactly would you evaluate an arithmetic expression like

\[ 3 + 6 \times 2.5^3 - \max(-3.4, 0) \times \cos(3.3) \]

You want to follow order of operations, of course, but how would you actually decide which operations to perform in which order? Truth be told, the way that humans do arithmetic isn’t so efficient, algorithmically speaking. Consider writing a program to do this. You might need to search every term of the arithmetic expression in order to find the one with highest precedence. Doing this over and over would be computationally expensive.

Now, imagine that you were an engineer at Texas Instruments or Apple, and you were tasked with the implementation of a calculator. Clearly, the naïve algorithm isn’t going to work here. What could you do instead?

This homework will walk you through an efficient algorithm for arithmetic computation that considers order of operations. We will first build a relevant data structure, then evaluate something called a “postfix” arithmetic expression, and finally convert a conventional “infix” arithmetic expression into this postfix form.

Part One: Implementing a Deque (25 points)

Files to submit: ResizingDeque.java, ResizingDequeTest.java

“Deque”, pronounced like “deck”, stands for double-ended queue, and generalizes both queues and stacks: you can add/enqueue (offer) and delete/dequeue (poll) from the structure at either end front or back. The usual queues only support enqueuing at the back and dequeuing at the front. Stacks support adding (push) and deleting (pop) at the front. Your deque should also support null entries as elements. You will be implementing the Java Deque interface. You can find the documentation here.
Note that ResizingDeque extends AbstractDeque. We have provided the abstract class AbstractDeque in order to reduce the amount of cruft you have to write to implement the Java Deque interface.

There are some points of difference between your expected ResizingDeque and the Java Deque, however. When implementing the ResizingDeque methods for add*, poll*, and offer*, you do not need to worry about throwing a ClassCastException. You also do not have to throw an IllegalStateException for the add* methods. Additionally, the add() method should always return true. Due to the resizing nature of your implementation, your data structure will have no notion of a maximum capacity.

Implementation: Resizing Array

You will implement a deque using array resizing, as discussed in class. Inside ResizingDeque, you should have an array for storing the elements of your deque. The array should start at 2 elements, and it should double in size (resize up) every time it fills up. That is, whenever the addition of a new element would cause the deque to exceed the size of the array. Whenever the removal of an element would cause the number of elements in the deque to fall to less than one quarter of the size of the array, you should cut the size of the array in half (resize down: don’t want to waste space!).

Your implementation will also want to use a concept known as wraparound. That is to say, after a series of pushes and pops, you may have a case where the head of your deque is not at array index 0, but maybe at an index farther along the array (like array index 4). After some more pushes, the index of the tail of your deque will get to the end of the array. The next push should then be at index 0, then index 1, etc.

Note that two of the unimplemented Deque methods in the AbstractDeque return Iterators. This homework assumes that you are familiar with iterators (textbook pp.138–9 as well as the Javadocs for the interface Iterator). For this homework, you do not need to implement remove for your iterators—in Java 8 remove is a default method that throws an UnsupportedOperation Exception. Moreover, since you cannot submit any additional files, your iterators must be private inner class(es). These two methods should return instances of the respective private inner class.

Note that Deque<E> is generic, so your implementation must also be generic; therefore the underlying array of elements must be of type E[]. Unfortunately, Java doesn’t cleanly support generic arrays, so the code to initialize a generic array of type E of size 2 in Java is

E[] elements = (E[]) new Object[2];

While your implementation must be generic, you should also be able to support adding and removing null elements to and from your data structure.

Note on loitering

When you implement any method that removes an element from the array (such as pollLast, removeFirstOccurrence, etc.), you should null out the associated entry in the array. Why should we do this? If we do not do this, then references to that element will “loiter” in the array. Java is a garbage-collected language. As long as we still have references to an object, then those references will prevent the reclamation of the memory used.
Part Two: Postfix Expression Calculator (20 points)

Files to submit: PostfixCalculator.java, PostfixCalculatorTest.java

In this section of the homework, you will implement a postfix calculator. It takes as input a postfix expression and evaluates it, returning a double-precision floating point value.

Postfix Expressions

Here is the recursive definition of “space-delimited” postfix expressions (PEs) as certain strings of characters:

- Any literal floating point constant (e.g., 17, 3.14, -2.71, -29, etc.) is a PE. We will also consider the “empty” input as a trivial postfix expression.
- If PE1 and PE2 are space-delimited postfix expressions and OP2 is a binary operator, specifically, one of +, *, /, -, `, max, min, then “PE1 PE2 OP2” is also a space-delimited postfix expression, where we may have one or more spaces between PE1 and PE2 and between PE2 and OP2.
- If PE is a space-delimited postfix expression and OP1 is a unary operator (for this homework, sin and cos), then “PE OP1” is also a space-delimited postfix expression where we have one or more spaces between PE and OP1.
- Only strings built by finitely many applications of the three rules above are space-delimited postfix expressions.

While the usual arithmetic expressions need precedence and associativity rules and/or parentheses for disambiguation, postfix expressions denote values unambiguously. As a result, there is a very simple left-to-right evaluation algorithm that uses a stack in which floating point numbers representing partial results are stored during the evaluation.

You will have to figure out by yourselves how to use the stack but the following evaluation traces will help:

- 5 -3.14 * 1 - → -15.70 1 - → -16.70 .
- 3 5 max 2 sin ^ → 5 2 sin ^ → 5 0.909297 ^ → 4.32087 .
- 1 2 3 + * → 1 5 * → 5 .

Implementation

You will need to implement the calculate(Iterator<String>) method in the PostfixCalculator class. We also provide a simple convenience method calculate(Iterable<String>), so that you can test your input with something along the lines of

```java
double result = PostfixCalculator.calculate(Arrays.asList("17", "17", "^"));
```
If your calculator receives an input that is not a valid postfix expression, such as \(2 \times 3\) and potentially many others, it should throw a `java.lang.IllegalArgumentException`.

Your implementation is not allowed to use the `Stack` class from the Java Collection Framework. Instead, you must implement the stub methods in the `DequeStack` class and use that class in your implementation. As the name obviously implies, you must use your `ResizingDeque` in your implementation of the `DequeStack`.

For parsing, we have provided a class `Tokens` that provides convenience methods for parsing individual token strings and `Operator` that enumerates all possible tokens for this assignment.

**Part Three: Infix to Postfix Translator (30 points)**

Files to submit: `ShuntingYard.java`, `ShuntingYardProcessorFactory.java`, `ShuntingYardTest.java`, `DequeStack.java`.

We now turn to arithmetic expressions written in the “usual” way, namely with (most) operators between operands (i.e., numbers) and with parentheses. Because of the position of operators we call the usual expressions **infix expressions**. In this part of the homework you will focus on the translation of infix expressions into postfix expressions. By composing this with the postfix calculator in part two, we can end up evaluating infix expressions. (FYI, see on p.129 of the textbook how to “merge” the idea of the algorithm in part three and the idea of the one in part two and thus produce an algorithm that evaluates infix expressions without an explicit intermediate translation into postfix.)

Your algorithm should handle infix expressions which are strings built from floating point constants (e.g., 5, 3.14, -2.71, -29, etc.), binary infix operators +, *, /, -, ^, the binary functions `max`, `min` and the unary functions `sin` and `cos`. In forming postfix expressions we have treated `max`, `min`, `sin`, `cos` as if they were arithmetic operators. In infix expressions, however, we use them for “function syntax”; that is, if \(E_1\) and \(E_2\) are infix expressions then the following are also infix expressions: `max ( E_1 , E_2 )` and `min ( E_1 , E_2 )` as well as `sin ( E_1 )` and `cos ( E_1 )`.

Here are some examples of infix expressions that use parentheses to disambiguate together with equivalent expressions that use precedence and associativity rules to disambiguate.

- \(( 5 \times -3.14 ) - 1 = 5 \times -3.14 - 1\)
- \(\text{max} ( 3 - 4 / \sin ( 2 ) , 5 ) = \text{max} ( 3 - ( 4 / \sin ( 2 ) ) , 5 )\)
- \(3 \times 6 \times 2.5 \times 4 \times 5 - \text{max} ( -3.4 , 0 ) \times \cos ( 3.3 ) = ( 3 + ( 6 \times ( 2.5 \times ( 4 \times 5 ) ) ) - ( \text{max} ( -3.4 , 0 ) \times \cos ( 3.3 ) )\)

**Caution:** When producing the translated output, you should use the original token strings provided in the input. Do not create your own token strings for the output. Our testers will check that the tokens match the input strings exactly, thus any tokens you generate that may represent the same literal (e.g. 3.0 v.s. 3) or operator will be reflected as a failure in the tests.
Shunting-yard

**Shunting-yard** is an algorithm invented by Edsger Dijkstra that converts expressions from infix notation to postfix notation. It is a stack-based algorithm, just like the evaluation of postfix in the previous task.

The central idea of the algorithm is to move tokens from the input to the output, “holding” operators in the stack until their operands have been moved, and then popping them from the stack to the output. Note that the stack is for operators only, not operands!

Here’s a brief sketch of the most basic version:

- Look at the first token.
- If it is a number, append it to the output.
- If it is an operator, then: while the stack is not empty and the precedence of the current operator is less than or equal to the one on the top of the stack, pop the top of the stack and append it to the output. Push the current operator onto the stack.
- Repeat with successive tokens until the input is exhausted.
- Then, pop the entire stack into the output.

Here are some hints for how the other pieces work:

- Parentheses—A left parentheses essentially acts as the lowest precedence operator, resetting the precedence of the stack (convince yourself of this). Consequently, the behavior of the right parentheses is to pop the stack into the output until the left parentheses is found. Recall that there are no parentheses in postfix expressions. So what happens next?
- Associativity—When an operator is left associative, the leftmost one is applied first. Thus, popping operators of equal precedence off the stack makes sense in the left associative case (again, convince yourself of this). Does this make sense in the right associative case, e.g., for the “operator?”
- Functions—What “precedence” does a prefix function in an infix expression have? Based on this “precedence”, figure out when to push function tokens onto the stack. The trick with functions is that the left parentheses can be treated like any other left parentheses. The different behavior comes in when a comma is encountered. Also, consider what should happen to the function token on the stack after the right parentheses has been found and processed.

*Note:* Shunting-yard operator precedence is *loosely* based on arithmetic order of operations. (From highest to lowest, arithmetic order of operations precedence is: functions, parentheses, exponents, multiplication and division, subtraction and addition). This brings us to reiterate a point hinted at in the above description of shunting-yard. Note also that the order of operations precedence from arithmetic is *not* necessarily identical to the operator precedence you want to use when implementing this algorithm. To be more specific, parentheses work differently when being read from the input token stream versus when being read from the stack.

The [Wikipedia article](https://en.wikipedia.org/wiki/Shunting-yard_algorithm) for shunting-yard also has a good coverage of this algorithm, and is a useful reference for implementing it.
Implementation

In the provided ShuntingYard stub, you have three versions of the function in the InfixToPostfix class, each one more complicated than the last. You should be able to re-use helper functions as well as much of the method body between these three iterations. In fact, you should begin the medium version with a copy of your code for the easy one, and similarly begin the hard version with a copy of your code from the medium one. Make sure each version is working correctly before you move on to the next\(^1\) Remember that the expected output for the shunting yard methods is an Iterable of token strings.

The new things you must implement (incrementally) in each method are:

- **Easy**—Numbers and left associative +, *, /, - operators
- **Medium**—Parentheses
- **Complete**—Right associative operators ^ and functions max, min, sin, cos.

You may assume that all inputs for the shunting-yard implementations are syntactically valid (i.e. 1 + * 2 will not be an input) and have been tokenized correctly. However, you must account for unmatched parentheses, such as 1 + ( 2, by throwing an UnmatchedParenthesesException and illegal tokens (tokens that are not valid operators or literals as determined by the Tokens methods) by throwing an UnknownTokenException.

Keep in mind that valid tokens for the latter shunting yard implementations could be invalid for the earlier ones.

Part Four: Asymptotic Analysis (10 points)

Files to submit: AnswersToDequeQuestions.java, AnswersToShuntingYardQuestions.java

You will thus answer some questions about the running time complexity of your code. The questions are specified in the interface DequeQuestions.java, and you should provide your answers by filling out the return values in AnswersToDequeQuestions.java. The following questions are also specified in the comments to each method, and you can answer each question through code using the RunningTimeAnalysisAnswer enum we've provided in the edu.upenn.cis121.hw5 package.

Consider a new implementation of Deque using linked lists as the underlying data structure, called LinkedListDeque. Notice that you should perform an amortized running time analysis of the ResizingDeque methods. (By amortized, you may assume that any given offer or poll won't resize the array, since it happens so infrequently.)

1. What is the running time of LinkedListDeque.contains()?  

---

\(^1\)As explained in the Javadocs of the ShuntingYard class, you don’t actually have to write all three. You just need to satisfy the specification of ShuntingYardProcessor.getInstance(Complexity). That means you could return a super shunting yard method that handles the highest complexity, since the lower shunting yard complexity levels are subsets of the higher shunting yard levels. However, make sure to keep in mind that, you should check for invalid tokens accurately, since a token valid for a more complex case may not be valid for a simpler one.
2. What is the running time of `LinkedListDeque.offerBack()`?
3. What is the running time of `LinkedListDeque.pollFront()`?
4. What is the running time of `ResizingDeque.contains()`?
5. What is the running time of `ResizingDeque.offerBack()`?
6. What is the running time of `ResizingDeque.pollFront()`?

The interface `ShuntingYardQuestions` has a few questions about the running time and functionality of the shunting-yard algorithm. These questions should be answered in the class `AnswersToShuntingYardQuestions`.

1. What is the running time of your infix to postfix (shunting-yard) algorithm based on a dynamic array implementation of a stack (as presented in class)?
2. What is the running time of your infix to postfix (shunting-yard) algorithm based on a linked list implementation of a stack?
3. (True/False) Given a valid input, the following stack can occur during the shunting-yard algorithm.
   
   * 
   +
   -

4. (True/False) Given a valid input, the following stack can occur during the shunting-yard algorithm.
   
   -
   -

Extra Credit: Push-Based Shunting-Yard (10 points)

Files to submit: `ShuntingYardPushHandler.java`, `ShuntingYardPushHandlerFactory.java`

In this optional part, you will implement the shunting-yard algorithm in a form similar to modern push-parsing.

There are two primary models: pull parsing and push parsing. Control flow and performance characteristics are drastically different.

Streaming pull parsing refers to a programming model in which a client application calls methods on a parsing library when it needs to interact with the input data; that is, the client only gets parsed data when it explicitly asks for it. The client controls the application thread, and calls methods on the parser when needed.

Streaming push parsing refers to a programming model in which a parser sends (pushes) data to the client as the parser encounters elements in the input; that is, the parser sends the data whether or not the

---

client is ready to use it at that time. The parser controls the application thread, and the client can only accept invocations from the parser.

Implementation

Please take a look at PushParsingShuntingYard, which is an extremely simple example of a push parser. You must fill out the class ShuntingYardPushHandler, which is a class implementing the interface PushParsingShuntingYard.ShuntingYardHandler, which hooks into specific points in PushParsingShuntingYard. The interface PushParsingShuntingYard.ShuntingYardHandler specifies events on which you can listen. The method PushParsingShuntingYard.ShuntingYardHandler.finish() completes the push parsing and returns the Iterable result.

Style and Tests (15 points)

The above parts together are worth a total of 85 points. The remaining 15 points are awarded for code style, documentation, and sensible tests. Style is worth 5 of the 15 points, and you will be graded according to the 121 style guide.

Also, please refer to the Java testing guide on the course website.