CSE 214 : Data Structures (Fall 2023) – Assignment II

Instructor: Dr. Ritwik Banerjee
Computer Science, Stony Brook University

This assignment comprises two parts and 10pages. In this assignment, you are free to use the `java.util.Stack` class readily available to you in Java. You should not, however, use any tree class provided by core Java (or any other external library). Remember to follow the development environment described in the syllabus. **This assignment is due by 11:59 pm, Oct 30 (Monday)**.

Note: whenever you are implementing a data structure corresponding to an abstract data type, you can write additional methods that are not required by the interface. These additional methods, however, are not a part of the data type's "contract", and therefore, must be private.

1. Arithmetic Expressions

In mathematics as well as the theory of programming languages, we often have to deal with strings with balanced parentheses, since they are a helpful construct in parsing and checking for valid mathematical expressions. We will use a slightly simplified version of this notion, and consider only the "(" and ")" parentheses. The idea is to check if an arithmetic expression is balanced as far as its parentheses are concerned. For example: `()`, `(3)`, and `2×((5+3)/2)` are balanced words, but `1-4(` and `((4+2)` are not. The empty string is, by default, considered to be balanced.

Some of the code for this class is already provided to you. All you need to do is complete the body of the method `isBalanced` in the `BalancedWord` class, shown below:

BalancedWord.java

```
public class BalancedWord {
  private final String word;

  public BalancedWord(String word) {
    if (isBalanced(word))
      this.word = word;
    else
      throw new IllegalArgumentException(String.format("%s is not a balanced word.", word));
  }

  private static boolean isBalanced(String word) { return false; } // TODO

  public String getWord() { return word; }
}
```

In general, just like verifying balanced parentheses, we would also like to verify whether a given arithmetic expression is a valid. For example, `(2-3+)` is a balanced word, but not a valid arithmetic expression. This task, however, is beyond the scope of this homework. For the purpose of this homework, you can safely assume that if an expression is balanced, it is also a valid arithmetic expression. The next part of this first task consists of two broad steps:

(i) Convert a given infix expression to its corresponding postfix expression, and
(ii) evaluate the postfix expression to obtain its final numeric value.

1.1. Convert infix to postfix

For this, you are being provided with an interface called `Converter`. It has a static nested class called `TokenBuilder`. Having a class defined inside an interface is perhaps new to you. It is implemented in this way so that *if*, in future, we want to add a different type of conversion (say, infix to prefix), we can use the same interface without having to worry about finding (or coding) the proper token builder[1].

The `TokenBuilder` class should be used to identify multi-digit as well as decimal numbers as single tokens (recall the discussion on tokens in the lectures). For this assignment, you can safely assume that at most two decimal places will be used to test your code. For example, $7.25 + 5/4$ is expected to be parsed and evaluated properly to 8.5, but you need not worry about handling higher precision numbers like 3.14159 in your test input. Similarly, you need only provide the final evaluation result rounded to two decimal places. For example, if the input expression is "355/113", your output must be 3.14.

Your task is to write the `ToPostfixConverter` class, which must implement the `Converter` interface. The conversion algorithm is provided to you in Appendix A. Rules to follow:

1. The infix to postfix conversion algorithm must be implemented in the `ToPostfixConverter.convert` method.
2. The `ToPostfixConverter` class must utilize the `Operator` enumerable type (*i.e.*, `enum`). Any use of hard-coded parentheses or operators like '+' in the code will carry score penalties.

Converter.java

```
/**
 * This interface defines the structure of any converter to be used for conversion of arithmetic
 * expressions between infix, prefix, and postfix types.
 *
 * @author Ritwik Banerjee
 */
public interface Converter {
    /**
     * The fundamental method of any class implementing this interface. It converts the given
     * arithmetic expression to another type, depending on the implementation.
     *
     * @param expression the given arithmetic expression
     */
    String convert(ArithmeticExpression expression);

    /**
     * Given a string and a specific index, returns the next token starting at that index.
     *
     * @param s     the given string
     * @param start the given index
     * @return the next token starting at the given index in the given string
     */
    String nextToken(String s, int start);

    /**
     * Determines whether or not a string is a valid operand.
     *
     * @param s the given string
     * @return <code>true</code> if the string is a valid operand, <code>false</code> otherwise
     */
    boolean isOperand(String s);

    /**
     * This class handles the parsing of tokens from a string. This is helpful in situations where
```

[1]This could have also been done by creating an abstract class implementing `Converter`, and then have the other classes extend that abstract class.

```
     * a single token may take up more than one character in the string.
     */
    class TokenBuilder {
        /**
         * The {@link StringBuilder} object used internally. This is used because {@link String}s
         * in Java are immutable, while we may want to build a token as we parse from left to
         * right one character at a time.
         */
        private StringBuilder tokenBuilder = new StringBuilder();

        /**
         * @see StringBuilder#append(char)
         */
        public void append(char c) {
            tokenBuilder.append(c);
        }

        /**
         * @return the final string object that represents a single token
         * @see StringBuilder#toString()
         */
        public String build() {
            return tokenBuilder.toString();
        }
    }
}
```

Operator.java

```
/**
 * This class provides the fixed enumerable types of operators to be used in the conversions and
 * evaluations of simple arithmetic expressions. Each operator is also equipped with a rank, which
 * serves to determine the precedence between two operators in case brackets are not enough to
 * resolve ambiguity of operation order.
 *
 * Since the variable and method names are obvious, documentation has not been provided for each
 * object or method individually.
 *
 * @author Ritwik Banerjee
 */
public enum Operator {

    // lower rank indicates higher priority or precedence
    LEFT_PARENTHESIS('(', 0),
    RIGHT_PARENTHESIS(')', 0),
    MULTIPLICATION('*', 1),
    DIVISION('/', 1),
    ADDITION('+', 2),
    SUBTRACTION('-', 2);

    private final char symbol;
    private final int  rank;

    Operator(char c, int rank) {
        this.symbol = c;
        this.rank   = rank;
    }
```

```
    public char getSymbol() { return symbol; }

    public int getRank() { return rank; }

    public static boolean isOperator(char c) {
        return c == ADDITION.symbol || c == SUBTRACTION.symbol ||
               c == MULTIPLICATION.symbol || c == DIVISION.symbol;
    }

    public static boolean isOperator(String c) { return isOperator(c.charAt(0)); }

    public static Operator of(char c) {
        if (c == LEFT_PARENTHESIS.symbol)
            return LEFT_PARENTHESIS;
        if (c == RIGHT_PARENTHESIS.symbol)
            return RIGHT_PARENTHESIS;
        if (c == MULTIPLICATION.symbol)
            return MULTIPLICATION;
        if (c == DIVISION.symbol)
            return DIVISION;
        if (c == ADDITION.symbol)
            return ADDITION;
        if (c == SUBTRACTION.symbol)
            return SUBTRACTION;
        throw new IllegalArgumentException(String.format("Invalid operator character %c.", c));
    }

    public static Operator of(String c) { return Operator.of(c.charAt(0)); }
}
```

1.2. Evaluating postfix expressions

Now that we have the code to obtain postfix expressions, we can evaluate them. In this part, you are required to mimic the design of the conversion process above:

1. You must create an interface called `Evaluator`. This is the abstract data type for evaluation, and the actual data structure will be a class implementing this interface. Just like the `Converter` interface has the `convert` method, this `Evaluator` interface must have the following:

   ```
   double evaluate(String expressionString);
   ```

   You may have additional methods as well, but that is entirely up to you.

2. Then, you must create a `PostfixEvaluator` class that implements this `Evaluator` interface. Here, too, we will mirror what we did in the conversion process. Rules to follow:
   (a) The postfix evaluation algorithm is given to you in Appendix B. This algorithm must be coded in the `PostfixEvaluator.evaluate` method.
   (b) The `PostfixEvaluator` class must utilize the `Operator` enumerable type. Hard-coded use of parentheses or operators like '+' in the code will carry score penalties.

For this first section of the assignment, a simple driver method is given to you as the `main(String[] args)` method in the `ArithmeticExpression` class. It requires an input file, where one infix expression is provided per line. Also pay close attention to how `try-catch` blocks are used here to handle different types of errors.

ArithmeticExpression.java

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

/**
 * This class defines an arithmetic expression as a wrapper around balanced words. It also
 * provides the <code>main</code> method to run the application to evaluate such expressions.
 *
 * @author Ritwik Banerjee
 */
public class ArithmeticExpression {
    /** Provide the full path to the input file. */
    private static final String INPUT_PATH = "";

    /** The balanced word around which we are wrapping. */
    private BalancedWord expression;

    /**
     * The constructor for this class, simply sets the balanced word based on the given expression.
     *
     * @param  expression  the given expression
     * @throws IllegalArgumentException if the given expression is not a valid balanced word
     */
    public ArithmeticExpression(String expression) throws IllegalArgumentException {
        this.expression = new BalancedWord(expression);
    }

    /**
     * @return the string representation of the balanced word
     */
    public String getExpression() { return expression.getWord(); }

    /**
     * The main method to run the application. It reads from the file specified by
     * {@link ArithmeticExpression#INPUT_PATH}, prints out the equivalent postfix expressions,
     * and then prints out the final evaluation as a <code>double</code>.
     */
    public static void main(String... args) {
        File input = new File(INPUT_PATH);
        try (BufferedReader bufferedReader = new BufferedReader(new FileReader(input))) {
            String line;
            while ((line = bufferedReader.readLine()) != null) {
                System.out.printf("Input: %s%n", line);
                try {
                    ArithmeticExpression a = new ArithmeticExpression(line.trim());

                    Converter converter = new ToPostfixConverter();
                    String    postfix   = converter.convert(a);
                    System.out.printf("\tPostfix: %s%n", postfix);

                    Evaluator evaluator = new PostfixEvaluator();
                    double    result    = evaluator.evaluate(postfix);
                    System.out.printf("\tValue: %f%n", result);
                    System.out.println();
                } catch (IllegalArgumentException e) {
```

```
                    System.out.println(e.getMessage());
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

2. Set abstract data type and implementation

The `Set` abstract data type stays true to its mathematical definition:

1. Duplicate elements cannot exist in a set.
2. Given any set S and element x, it can be determined whether $x \in S$ (read as "x is a member of S").

As computer scientists, however, we are interested in *dynamic sets*, which can grow and shrink through the addition and removal of elements. Further, we also want to check for duplicates and set membership as fast as possible. In this second part of the assignment, you are required to work towards this only for the `Integer` data type. Note that we are talking about the Java object `Integer`, and not the primitive `int`.

First, to define the abstract data type, create an interface called `DynamicSet`. This interface must have the following methods:

Method declarations in DynamicSet.java (interface):

```
/**
 * @return the number of elements in this set
 */
int size();

/**
 * Checks if a specified element is a member of this set.
 *
 * @param x the specified element to be checked for set membership
 * @return <code>true</code> if and only if this set contains the specified element
 */
boolean contains(Integer x);

/**
 * Adds a specified element to this set. Addition is successful unless the set already contains
 * this element.
 *
 * @param x the specified element to be added to this set
 * @return <code>true</code> if and only if the specified element was successfully added
 */
boolean add(Integer x);

/**
 * Removes a specified element from this set.
 *
 * @param x the specified element to be removed from this set
 * @return <code>true</code> if and only if this element was successfully removed
 */
boolean remove(Integer x);
```

Now, you must write a class called `DynamicIntegerSet`, which implements the above interface. Internally, this class must use a binary search tree to store the set elements. Just like linked lists, we are going to use

nodes for this. The abstract data type for such a node is given to you below:

```
public interface PrintableNode {
    /**
     * @return the value of this node as a string
     */
    String getValueAsString();

    /**
     * @return the left child node of this node
     */
    PrintableNode getLeft();

    /**
     * @return the right child node of this node
     */
    PrintableNode getRight();
}
```

Your `DynamicIntegerSet` must be a binary search tree that uses the above type of nodes. In other words, there must be a nested static class as follows:

```
public class DynamicIntegerSet implements DynamicSet {
  public static class Node implements PrintableNode {
    Integer  data;
    Node left, right;

    Node(int x) { this(x, null, null); }

    Node(int x, Node left, Node right) {
        this.data  = x;
        this.left  = left;
        this.right = right;
    }

    @Override
    public String getValueAsString() { return data.toString(); }

    @Override
    public PrintableNode getLeft() { return left; }

    @Override
    public PrintableNode getRight() { return right; }
  }

  // this method must be there exactly in this form
  public Node root() { return this.root; }

  // rest of your code for this class, including the size, contains, add, and remove methods
```

Rubric

Arithmetic expressions (50 points)

1. Complete the `isBalanced` method in the `BalancedWord` class. (10)

2. Convert infix to postfix through the `ToPostFixCoverter.convert` method. (18)
3. Define the `Evaluator` interface as described in this document. (4)
4. Evaluate arithmetic expressions through the `PostFixEvaluator.evaluate` method. (18)

Set abstract data type and implementation (50 points)

5. `DynamicIntegerSet` implements the `DynamicSet` interface, as provided in this document. (5)
6. The `size()` method correctly returns the number of elements in the set (5)
7. The `add(Integer x)` method adds an element to the dynamic set (*i.e.*, to the binary search tree) using the proper insertion algorithm for a binary search tree, which takes $\theta(\log_2 n)$ time on average. (15)
8. The `remove(Integer x)` method removes an element from the dynamic set (*i.e.*, from the binary search tree) using the proper deletion algorithm for a binary search tree, which takes $\theta(\log_2 n)$ time on average. (15)
9. The `contains(Integer x)` method searches for an element in the dynamic set (*i.e.*, in the binary search tree) using the proper binary search algorithm, which takes $\theta(\log_2 n)$ time on average. (10)

Very important notes:

1. The correctness of your code will be tested with various test cases, and NOT by reading through your code. In lectures, we have discussed the role of a client code. You should think of the grading code as a client code. For example, the grader will test the `size()` method by creating a binary search tree using your `add(Integer x)` method a few times, and then calling the `size()` method. The grader will not read your code line-by-line to assess its correctness!
2. Follow the driver `main` method given to you for the first part. This method is given to you so that you can test your own code exactly as it will be tested by the grading process.
3. The `Node` must implement the `PrintableNode` interface, and your dynamic integer set class must contain the `root()` method exactly as provided. Any changes here will prevent the grader from printing the tree structure (and they will not be able to verify the correctness of your code). On the course web page, a method called `PrintTree(PrintableNode node)` is provided. You don't have to do anything with this method, but you may use it to check if your tree is printed properly. You may also use it to verify the correctness of your code. This method is, essentially, what the grader will also use to check your implementation. Here is a sample output for a very simple binary search tree:

```
          7
          |
      ------------
     |            |
     6            11
     |
  -----
  |
  5
```

4. Keep in mind that if your submitted code doesn't compile, it cannot be graded! It is better to submit code where *some* methods work properly and can be graded by the driver method (or, for the second part, the tree printing method) than to submit a lot of code that doesn't compile or doesn't abide by the provided guidelines.

Appendix A: Infix-to-postfix conversion algorithm

This algorithm operates under the assumption that the usual priority order of arithmetic operators holds, and that the input expression is a valid infix expression. It converts infix expressions like `a+b*c+(d*e+f)*g` into its equivalent postfix `a b c * + d e * f + g * +`.

Note the single space between tokens in the postfix notation. This is necessary so that we can distinguish between numeric operands. For example, "142/" is 14 divided by 2, but "142/" is ambiguous. Your postfix expressions must incorporate this inter-token space.

Algorithm

1. Initialize an empty stack.
2. When an operand is read, it is immediately placed on the output.
3. When a left parenthesis is read, push it onto the stack.
4. When a right parenthesis is read, pop the stack and keep writing the tokens to the output until a left parenthesis is seen. This left parenthesis is popped, but not written to the output. The original right parenthesis is not written to the output either.
5. When an operator is read:
   (a) If the stack is empty, or its top element is a left parenthesis, push the input operator onto the stack.
   (b) If the input operator has higher precedence than the top of the stack, push the input operator onto the stack.
   (c) If the input operator has the same precedence as the top of the stack, use the associative rule from algebra: pop and add the top of the stack to the output, and then push the input onto the stack.
   (d) If the input operator has lower precedence than the top of the stack, pop the stack and add the popped element to the output. Then, test the input operator against the new top element.
6. At the end of the input expression, pop and add each token to the output one by one (there should not be any remaining parentheses).

Appendix B: Evaluating postfix expressions

The postfix notation (also known as the *reverse Polish notation*) can be very easily evaluated with the aid of a stack.

Algorithm

1. Initialize an empty stack.
2. When an operand is seen, push it onto the stack.
3. When an operator is seen, pop two elements successively from the stack, evaluate the expression (this will always make sense, since we are only dealing with binary operators) and push the result back onto the stack.
4. When there is a single operand left on the stack, pop and return it as the final result. If the input is valid, the stack should have exactly one operand left when the expression has been completely read.

Recall from the lectures that the postfix notation does not need any operator priorities or parentheses. Algorithmically, it is much easier to compute based on this, rather than the infix notation.

Example

Input: Postfix expression: 6 5 2 3 + 8 * + /

1. The first four operands are pushed onto the stack [6, 5, 2, 3] ← this end is the top
2. Then, '+' is read. So 2 and 3 are popped, $2+3$ is evaluated to be 5, and this result is pushed onto the stack. The stack now holds [6, 5, 5].
3. Then, 8 is pushed onto the stack. [6, 5, 5, 8].

4. Then '*' is read, so 5 and 8 are popped. The result$5 \times 8 = 40$ is pushed onto the stack. So the stack now holds [6, 5, 40].
5. Then, '+' is read. So 5 and 40 are popped, and the result 45 is pushed onto the stack, which now holds [6, 45].
6. Next, '/' is read. So two elements are popped. The result of '6/45' is evaluated, which turns out to be 0.13 (remember, this assignment requires you to provide at most two decimal places of numeric precision). This result is pushed onto the stack.
7. Now, the stack has only one operand and we have finished reading the input expression. So pop this single operand as the final result.