HW3

Due: Tue. Jun. 9th by 11:55 pm

This assignment will be marked out of 30 points. You also have an opportunity to earn 5 bonus points.

- *This is an individual assignment. Please review the rules on academic misconduct on the course D2L page.*
- *DO NOT share your answers/code with others in any way (in person or electronically); DO NOT ask others to provide you with answers/code; write up the solutions/code yourself and clearly cite any external sources (books, websites, forums, discussions with peers etc.) that helped you arrive at your solutions.*
- *When asked to implement something, the implementation must be your own.*
- If you have any doubts, please contact the instructor.

This assignment consists of two parts. The first part consists of problems that will give you more practice working with binary trees. The second part consists of a coding exercise that explores the use of trees in parsing and evaluating arithmetic expressions.

Part I - Written

1. (5 points)
   Prove the following:
   - A *binary* tree with n nodes has $n + 1$ `null` links.
   - In a non-empty binary tree, the number of full nodes plus one is equal to the number of leaves. (A *full node* is a node that has two children.)
2. (5 points)
   Design a linear time algorithm to test whether a binary tree is a binary search tree. Provide pseudocode or Java code for your algorithm and justify that your algorithm is linear in the number of nodes.
3. (5 points)
   Analyze the worst case and best case running times of inserting n elements into an initially empty binary search tree. For each case, provide details of your analysis starting from counting relevant operations to an accurate asymptotic characterization.

Submission

Please handwrite or typeset your solutions to these problems. If hand-writing, please write neatly and make sure that your responses are clearly visible in your scanned/photographed images.

Please submit via gradescope to the assignment entitled 'HW3_Written'. Gradescope will accept both PDF and image file formats. Allow yourself plenty of time before the deadline to ensure a smooth submission process.

Part II - Coding

Write a parser that parses an arithmetic expression given in *infix* notation into a binary expression tree. Your parser should support infix expressions consisting of binary operators {`'+'`, `'-'`, `'*'`, `'/'` }, numbers (in integer format including both positive and negative integers), as well as parentheses (which may be nested). Once an expression has been parsed into an expression tree, your program will additionally be able to perform preorder and postorder traversals on the tree, evaluate the expression, and simplify the tree.

You can perform the parsing either using a stack (worth 15 points) or via recursive descent (worth 15 points + 5 bonus) using an expression grammar for infix expressions.

Stack-based Parsing (15 points)

In class, we discussed an algorithm for evaluating postfix expressions. We can easily adapt that algorithm to build an expression tree. The resulting algorithm is shown below. It assumes that there is a routine called ExpTreeNode(s, T_1, T_2), that builds an expression tree node with s as the element stored in the node, T_1 as the left child, and T_2 as the right child.

Data: Expression E in postfix notation.
Result: Expression tree T corresponding to E or error if a valid expression tree cannot be built from the input.

```
initialize stack S;
while there are more tokens in E do
    t ← next token;
    if t is a number then
        T ← ExpTreeNode(t, ∅, ∅);
        push T onto stack S;
    else
        // t is an operator
        if there are fewer than 2 nodes on S then
            return error;
        else
            T2 ← pop(S);
            T1 ← pop(S);
            T ← ExpTreeNode(t, T1, T2);
            push T onto stack S;
        end
    end
end
if there is only one node in S then
    T ← pop(S);
    return T;
else
    return error;
end
```

Note that in order to use this algorithm, you will first need to apply Dijkstra's shunting yard algorithm (refer to class slides) to convert the given infix expression to postfix.

- You may use the `Stack` class from the Java Collections Framework.

- Implementation of Dijkstra's shunting yard algorithm and the subsequent stack-based parsing algorithm must be your own. You are also *required* to use the provided `ExpressionTree` and `ExpressionTreeNode` classes (see below).
- Your implementation should check for errors and throw an exception if there are any syntax problems in the input expression.

Recursive Descent (15 + 5 points)

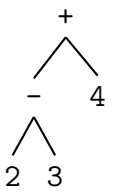
An expression grammar for infix expressions is given below.

```
E --> T { ( "+" | "-" ) T }
T --> F { ( "*" | "/" ) F }
F --> num | "(" E ")"
```

This grammar generates expressions consisting of the binary operators `+`, `-`, `*` and `/` as well as parenthetical expressions.

Notation: In the above grammar, braces `{}` are used to indicate productions that may repeat 0 or more times, `|` separate alternatives, while parentheses `()` are used for grouping. The only terminal symbol is a `num`. *For this assignment, you can assume that* `num` $\in \mathbb{Z}$.

Note that `*` and `/` have higher precedence compared to `+` and `-`. Operators with the same precedence are associated left-to-right (left-associativity), e.g. the expression `"2 - 3 + 4"` yields the following tree.



Before proceeding with your implementation, please make sure you understand the rules of the grammar and know how to apply them for parsing via recursive descent. For more information on recursive descent, please consult the resources posted on D2L.

Implementation

Write a Java class called `ExpressionTree` that implements the `ExpressionTreeInterface` interface that consists of the following methods.

```
public interface ExpressionTreeInterface {

  /**
   * Build a parse tree from an expression.
   * @param line String containing the expression
   *
   * @throws ParseException If an error occurs during parsing.
   */
  public void parse( String line ) throws ParseException;

  /**
   * Evaluate the expression tree and return the integer result. An
   * empty tree returns 0.
   */
```

```
  public int evaluate();

  /**
   * Simplifies this tree to a specified height h (h >= 0). After
   * simplification, the tree has a height of h. Any subtree rooted
   * at a height of h is replaced by a leaf node containing the
   * evaluated result of that subtree.
   * If the height of the tree is already less than the specified
   * height, the tree is unaffected.
   *
   * @param h The height to simplify the tree to.
   */
  public void simplify(int h);

  /**
   * Returns a parentheses-free postfix representation of the
   * expression. Tokens are separated by whitespace. An empty tree
   * returns a zero length string.
   */
  public String postfix();

  /**
   * Returns a parentheses-free prefix representation of the
   * expression. Tokens are separated by whitespace. An empty tree
   * returns a zero length string.
   */
  public String prefix();
}
```

The following files are provided to you on D2L:

1. `ExpressionTreeNode.java` - Represents a node in a binary expression tree.
2. `ExpressionTreeInterface.java` - The interface shown above.
3. `ExpressionTree.java` - A class skeleton which consists of method stubs that you are required to implement.

Additional Requirements

Include as many private methods as necessary to parse a given expression. All helper methods and/or nested classes should reside in the file `ExpressionTree.java`. DO NOT modify any of the other files.

You can assume that properly formatted input expressions will have tokens separated by whitespace. However, you should check for expressions that have syntax issues. If an input expression contains a syntax error, the `parse` method should throw a `ParseException`. In the thrown exception, include a description of the nature of the error.

When converting expressions to prefix or postfix, please separate tokens using whitespace. Prefix and postfix expressions should be parentheses-free.

Documentation and Testing

For all private methods that you add, please provide Javadoc comments to explain what the methods are doing.

Write a program to test your parser. Test with both short and long expressions as well as expressions with mixed operators, nested parentheses etc. A test program that is compatible with your implementation will be made available closer to the due date.

Submission and Grading

Your implementation will be tested by an autograder on Gradescope. When you submit, the autograder will run some unit tests and grade your submission based on functionality.

Please pay attention to the packaging structure of the files that are provided to you. Do not alter this structure or you will run into issues with the autograder on Gradescope.

- Please submit via Gradescope to the assignment entitled 'HW3_Coding'.
- *Please only submit the file* `ExpressionTree.java` *ensuring that the package declaration* `hw3.student` *is intact. Submit the file directly, do not compress it into an archive.*
- Please include any citations as comments at the the top of the file `ExpressionTree.java`.
- Once submitted, the autograder will compile and test your implementation providing you with output that shows the test results.
- You may submit multiple times. Only the most recent submission will be kept.
- You will be graded primarily based on functionality as tested by the autograder. The auto-graded score is subject to adjustments at the discretion of the marker.