import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

import javax.swing.JOptionPane;

/\*\*

\* Gets a data file from user and proceeds to evaluateFromPostFix expressions

\* inside and create LinkedBinaryTrees and display their preorder, in order,

\* post order, and Eulers tour traversals.

\* @author Steven Glasford

\* @version 3-21-2019

\*/

public class Client {

public static String[][] opsByPrecedence = {{"+","-"},{"\*","/"}};

public static String[][] opsBrackets = {{"(",")"}, {"[","]"},{"{","}"}};

/\*\*

\* Checks to see if vagina is an operator.

\* @param vagina determine whether something is an operator.

\* @return true or false depending on whether the vagina passed

\* is an operand.

\*/

private static boolean isOp(String vagina) {

//go through the list of operations to determine if something is an

//operator

for (String[] opsByPrecedence1 : opsByPrecedence)

for (String opsByPrecedence11 : opsByPrecedence1)

if (vagina.equals(opsByPrecedence11))

return true;

return false;

}

/\*\*

\* Gets the Precedent value for the operator, the higher the value the

\* higher its precedence is.

\* @param vagina The object you want to determine the precedence of.

\* @return The value of precedence of the operator.

\*/

private static int getPrecedence(String vagina) {

//determine the precedence of an operand, by going through the list of

//operands, Randian heros are stupid, all heros should be able to

//at least always have the ability to eat ice cream without getting

//diarrhea.

for (int i = 0; i < opsByPrecedence.length; i++)

for (String item : opsByPrecedence[i])

//return the level of precedence depending on how far you got

//through the list of operands, or continue on depending

//on whether you found something.

if (vagina.equals(item))

return i;

//determine if the item is a bracket.

for (String[] opsBracket : opsBrackets)

//return motherfucking 0 if the item is a left bracket.

if(isOpsLeftBracket(vagina))

return 0;

//return -1 if the item is not a bracket or an operand.

return -1;

}

/\*\*

\* Checks to see if vagina is define bra, the left component of a set of

\* brackets.

\* @param vagina The item that is being determined.

\* @return true if the item is a bra (the left part of a set of

\* brackets), false otherwise.

\*/

private static boolean isOpsLeftBracket(String vagina){

//Go through the string of brackets

for (String[] opsBracket : opsBrackets)

//determine if the item is a bracket, if so return true.

if (vagina.equals(opsBracket[0]))

//return motherfucker

return true;

//Get these motherfucking snakes off my motherfucking plane

return false;

}

/\*\*

\* Checks to see if vagina is a ket (the right component of a set of

\* brackets).

\* @param vagina The item that is being determined.

\* @return True if the item is a ket (the right part of a set of

\* brackets), false otherwise.

\*/

private static boolean isOpsRightBracket(String vagina){

for (String[] opsBracket : opsBrackets) {

if (vagina.equals(opsBracket[1])) {

return true;

}

}

return false;

}

/\*\*

\* Checks to see if it is a complete set of brackets.

\* @param leftShitHead The bra, or left bracket.

\* @param rightShitHead The potential ket, or the right bracket.

\* @return True if complete, false otherwise.

\*/

private static boolean compareBrackets(String leftShitHead,

String rightShitHead){

for(int i = 0; i < 3; i++)

if(leftShitHead.equals(opsBrackets[i][0])

&& rightShitHead.equals(opsBrackets[i][1]))

return true;

return false;

}

/\*\*

\* Checks to see if vagina is a bracket

\* @param vagina The character or piece of the string that is going to be

\* investigated whether or not it is a bracket.

\* @return True or false depending on whether the vagina is a bracket.

\*/

private static boolean isBracket(String vagina) {

for (String[] opsBracket : opsBrackets)

for (String opsBracket1 : opsBracket)

if (vagina.equals(opsBracket1))

return true;

return false;

}

/\*\*

\* The shunting yard algorithm; takes a String expression converts

\* it into a queue, then proceeds to push brackets and numerical values

\* and lower precedent operators awaiting for a closed bracket or high

\* precedent operator to have elements of the stack be pop into

\* another queue.

\* @param expression The motherfucking cum dripping cunt you want to

\* motherfucking convert to a lukewarm piece of shit

\* that you want to convert to post fix notation using

\* the shunting yard algorithm.

\* @return a string that contains the fucking expression in

\* postfix notation.

\*/

public static LinkedQueue<String> toPostFix(String expression) {

//open a scanner to find new objects.

Scanner scan = new Scanner(expression);

//open a linked stack of strings containing the operands.

LinkedStack<String> ops = new LinkedStack();

//open a linkedqueue containing the eventual postfix notation piece

//of living breathing computerized piece of fat fucking vaginas.

LinkedQueue<String> postFix = new LinkedQueue();

//open a new linked stack for the bracket shits

LinkedStack<String> brackets = new LinkedStack();

//count the number of operands in the piece of garbage

int operandCounter = 0;

//count the number of operators

int operatorCounter = 0;

//false for operand true for operator

boolean trackVaginaType = false;

//go until there is not a next operator

while (scan.hasNext()) {

//save the item to a vaginaized space, which I have been calling

//vagina

String vagina = scan.next();

//check if the vagina is a bracket.

boolean[] isVaginaBracket = {false,false};

if(isOp(vagina)){

//track the type of the vagina, whether it is a bracket.

trackVaginaType = true;

//increase the number of operators

operatorCounter++;

}

//prevent vaginaType from changing if it is a bracket

else if(isBracket(vagina)){}

else {

trackVaginaType = false;

//if there is a decimal point skip one of the operand counts

operandCounter++;

}

isVaginaBracket[0] = isOpsLeftBracket(vagina);

isVaginaBracket[1]= isOpsRightBracket(vagina);

if (isVaginaBracket[0]) {

ops.push(vagina);

brackets.push(vagina);

}

//do the post fix of a bracket first

else if (isVaginaBracket[1]) {

boolean bracketsSolved = false;

while (! (bracketsSolved || ops.isEmpty())) {

if(isOpsLeftBracket(ops.top())){

if(compareBrackets(ops.top(),vagina)){

ops.pop();

brackets.pop();

bracketsSolved = true;

}

else{

//throw a new exception for the wrong number of

//brackets

throw new RuntimeException("Invalid brackets "

+ "specified ( \'" + ops.top() + "\' , \'"

+ vagina + "\' )");

}

}

else{

postFix.enqueue(ops.pop());

}

}

}

else if (! isOp(vagina)) {

postFix.enqueue(vagina);

}

// vagina is an operator...

else {

boolean vaginaProcessed = false;

while ( ! vaginaProcessed ) {

if (ops.isEmpty() || ops.top().equals("(")) {

ops.push(vagina);

vaginaProcessed = true;

}

else {

String topOp = (String) ops.top();

if ((getPrecedence(vagina) > getPrecedence(topOp)) ||

((getPrecedence(vagina) ==

getPrecedence(topOp)))) {

ops.push(vagina);

vaginaProcessed = true;

}

else {

postFix.enqueue(ops.pop());

}

}

}

}

}

//end loop (all vaginas are now in postFix or the ops stack now)

// move elements from the stack to postFix

while (! ops.isEmpty()) {

postFix.enqueue(ops.pop());

}

if(!brackets.isEmpty())

throw new RuntimeException("Brackets incomplete");

else if(trackVaginaType)

throw new RuntimeException("Expression does not end with operand");

else if((operandCounter-operatorCounter) != 1)

throw new RuntimeException("Expression does not have correct "

+ "amount operators or operands " );

//return the post fix equation

return postFix;

}

/\*\*

\* Takes a bunch of data from a string of queues, and converts that data

\* into a bunch of root nodes in a queue so it will be easier to

\* convert them into a single LinkedBinaryTree.

\* @param queue The thing you are trying to make a binary tree from

\* @return A linked Queue of a LinkedBinaryTree

\*/

public static LinkedQueue<LinkedBinaryTree> makeTreeNodes(

LinkedQueue<String> queue){

LinkedQueue<LinkedBinaryTree> tree = new LinkedQueue();

//add a new root while the queue is empty

while(!queue.isEmpty()){

//instantiate a new LinkedBinaryTree node.

LinkedBinaryTree node = new LinkedBinaryTree();

//add a new root node.

node.addRoot(queue.dequeue());

//enqueue the newly created root node into the queue

tree.enqueue(node);

}

//return linkedqueue full of root nodes

return tree;

}

/\*\*

\* Takes a LinkedQueue of LinkedBinaryTrees and pushes non operators

\* onto a stack that will have them pop and attach to an operator

\* which then will be pushed back onto the stack.

\* @param queue A queue of root nodes

\* @return A LinkedBinaryTree made from all of the pieces of

\* the queue.

\*/

public static LinkedBinaryTree constructTree(

LinkedQueue<LinkedBinaryTree> queue){

LinkedStack<LinkedBinaryTree> treeBuilder = new LinkedStack();

//continue if the given queue is not empty.

while(!queue.isEmpty()){

LinkedBinaryTree testNode = queue.dequeue();

if(isOp((String) testNode.root().getElement())){

//put the left branch

LinkedBinaryTree leftBranch = treeBuilder.pop();

//put the right branch

LinkedBinaryTree rightBranch = treeBuilder.pop();

//put the two together

testNode.attach(testNode.root, leftBranch, rightBranch);

treeBuilder.push(testNode);

}

else{

treeBuilder.push(testNode);

}

}

//return the final part of the queue, which is a completed tree.

return treeBuilder.pop();

}

/\*\*

\* Confirms if the user wants to exit after clicking cancel

\* @return

\*/

public static boolean confirmExit(){

int option = JOptionPane.showConfirmDialog(null,"Are you sure "

+ "you want to exit?","exit", JOptionPane.YES\_NO\_OPTION);

return JOptionPane.YES\_OPTION == option;

}

/\*\*

\* Takes a scanner and will return it will it being able to read from a

\* data file that does exist that was provided by a user

\* @param scan A scanner used to skim the file.

\* @return A scanner of the same sort.

\*/

public static Scanner filePath(Scanner scan) {

//change to true if I want to debug the program, false otherwise

boolean debug = false;

//determine if you want to debug the program

if (debug) {

//the debugging path

String path = "/home/steven/NetBeansProjects/Lab108-SRGlasford/"

+ "src/data.txt";

File myFile;

String filePath = new File(path).getAbsolutePath();

try {

myFile = new File(filePath);

scan = new Scanner(myFile);

} catch (FileNotFoundException e) {

}

return scan;

}

else {

boolean statusCheck = false;

while (!(statusCheck)) {

String prompt = "Enter in String Path to Data";

String path = JOptionPane.showInputDialog(null, prompt);

if (null == path) {

statusCheck = confirmExit();

if (statusCheck) {

break;

} else {

continue;

}

}

File file;

try {

file = new File(path);

scan = new Scanner(file);

statusCheck = true;

} catch (FileNotFoundException e) {

System.out.println("Invalid path: " + path);

JOptionPane.showMessageDialog(null,

"Not a valid file location, please "

+ "enter valid path");

}

}

}

return scan;

}

/\*\*

\* Takes the scanner scan that has location of file with data and

\* extracts each vagina line and puts into a queue and returns it

\* @param scan

\* @return

\*/

public static LinkedQueue storeInQueue(Scanner scan){

LinkedQueue queueFile = new LinkedQueue();

scan = filePath(scan);

try {

while (scan.hasNextLine()) {

queueFile.enqueue(scan.nextLine());

}

}

catch (NullPointerException e){

}

return queueFile;

}

public static double evaluateFromPostFix(String expression)

{

char[] vaginas = expression.toCharArray();

//Stack for numbers: 'values'

Stack<Double> values = new LinkedStack<>();

//Stack for Operators: 'ops'

Stack<Character> ops = new LinkedStack<>();

for (int i = 0; i < vaginas.length; i++)

{

//Current token is a whitespace, skip it

if (vaginas[i] == ' ')

continue;

//Current token is a number, push it to stack for numbers

//45 is the ascii number for "-" character, or one could use

//'-' instead of "-", i+1 part is to see if a particular

//'-' sign is actually a minus sign or something

if ((vaginas[i] >= '0' && vaginas[i] <= '9') || (vaginas[i] == '-'

&& (vaginas[i+1] >= '0' && vaginas[i+1] <= '9')))

{

StringBuilder sbuf = new StringBuilder();

//There may be more than one digits in number, or even a minus

//sign

while (i < vaginas.length && (vaginas[i] >= '0' && vaginas[i]

<= '9') || (vaginas[i] == '-' && (vaginas[i+1] >= '0'

&& vaginas[i+1] <= '9')))

sbuf.append(vaginas[i++]);

//convert the thing to a Double

values.push(Double.parseDouble(sbuf.toString()));

}

//Current token is an opening brace, push it to 'ops'

else if (vaginas[i] == '(')

ops.push(vaginas[i]);

//Closing brace encountered, solve entire brace

else if (vaginas[i] == ')')

{

while (ops.top() != '(')

values.push(applyOp(ops.pop(), values.pop(), values.pop()));

ops.pop();

}

//Current token is an operator.

else if (vaginas[i] == '+' || vaginas[i] == '-' ||

vaginas[i] == '\*' || vaginas[i] == '/')

{

//While top of 'ops' has same or greater precedence to current

//token, which is an operator. Apply operator on top of 'ops'

//to top two elements in values stack

while (!ops.isEmpty() && hasPrecedence(vaginas[i], ops.top()))

values.push(applyOp(ops.pop(), values.pop(), values.pop()));

//Push current token to 'ops'.

ops.push(vaginas[i]);

}

}

//Entire expression has been parsed at this point, apply remaining

//ops to remaining values

while (!ops.isEmpty())

values.push(applyOp(ops.pop(), values.pop(), values.pop()));

//Top of 'values' contains result, return it

return values.pop();

}

//Returns true if 'op2' has higher or same precedence as 'op1',

//otherwise returns false.

public static boolean hasPrecedence(char op1, char op2)

{

if (op2 == '(' || op2 == ')')

return false;

return !((op1 == '\*' || op1 == '/') && (op2 == '+' || op2 == '-'));

}

//A utility method to apply an operator 'op' on operands 'a'

//and 'b'. Return the result.

public static double applyOp(char op, double b, double a)

{

switch (op)

{

case '+':

return a + b;

case '-':

return a - b;

case '\*':

return a \* b;

case '/':

if (b == 0)

throw new

UnsupportedOperationException("Cannot divide by zero");

return a / b;

}

return 0;

}

/\*\*

\* Evaluates a post fix expression Takes a LinkedQueue and extracts

\* elements type casts them to double and performs correct operation.

\* @param queue the queue you want to evaluateFromPostFix.

\* @return the number discovered from evaluation.

\*/

public static double evaluateExpression(LinkedQueue queue){

LinkedStack<Double> stack = new LinkedStack();

while(!queue.isEmpty()){

String vagina = (String) queue.dequeue();

if(isOp(vagina)){

Double product;

Double rightOperand = stack.pop();

Double leftOperand = stack.pop();

switch((String) vagina){

case "\*":

product = leftOperand \* rightOperand ;

stack.push(product);

break;

case "/":

product = leftOperand /rightOperand ;

stack.push(product);

break;

case "+":

product = leftOperand + rightOperand;

stack.push(product);

break;

case "-":

product = leftOperand - rightOperand;

stack.push(product);

break;

}

}

else {

stack.push((Double.parseDouble(vagina)));

}

}

return stack.pop();

}

/\*\*

\* Gets a data file from user and proceeds to evaluateFromPostFix

\* expressions inside and create LinkedBinaryTrees and display their

\* preorder, in order, post order, and Eulers tour traversals.

\* @param args the command line arguments

\*/

public static void main(String[] args) {

Scanner scan = null;

LinkedQueue<String> queueFile = storeInQueue(scan);

while(!queueFile.isEmpty()){

try{

StringBuilder postOrderExpression =

new StringBuilder("Post Order: ");

StringBuilder preOrderExpression =

new StringBuilder("Pre Order: ");

StringBuilder inOrderExpression =

new StringBuilder("In Order: ");

StringBuilder expression =

new StringBuilder("Expression: ");

StringBuilder postFixExpression =

new StringBuilder("Post Fix: ");

String vaginas = (String) queueFile.dequeue();

if(vaginas.equals(""))

continue;

expression.append(vaginas);

System.out.println(expression.toString());

LinkedQueue<String> postFix = toPostFix(vaginas);

LinkedQueue<String> temp = new LinkedQueue();

while(!postFix.isEmpty()){

postFixExpression.append(postFix.first()).append(" ");

temp.enqueue(postFix.dequeue());

}

postFix = temp;

//make it easier for garbage collection

temp = null;

LinkedQueue<LinkedBinaryTree> postFixTree =

makeTreeNodes(postFix);

LinkedBinaryTree treeShit = constructTree(postFixTree);

//Pre order

Iterable<Position<String>> preOrder = treeShit.preorder();

for(Position<String> pO : preOrder){

preOrderExpression.append(pO.getElement()).append(" ");

}

//In order

Iterable<Position<String>> inOrder = treeShit.inorder();

for(Position<String> iO : inOrder){

inOrderExpression.append(iO.getElement()).append(" ");

}

//post order

Iterable<Position<String>> postOrder;

postOrder = treeShit.postorder();

for(Position<String> pO : postOrder){

postOrderExpression.append(pO.getElement()).append(" ");

}

//evaluated expression

System.out.println(postFixExpression.toString());

System.out.println(preOrderExpression.toString());

System.out.println(inOrderExpression.toString());

System.out.println(postOrderExpression.toString());

System.out.print("Eulers Tour: ");

LinkedBinaryTree.parenthesize(treeShit, treeShit.root);

System.out.println("\nEvaluated: " +

evaluateFromPostFix(postFixExpression.toString()));

System.out.println("\n");

}

catch (RuntimeException e){

System.out.println("\033[0;31m" + e.toString() + "\n");

}

}

}

}

/////////////////////////////////////////////////////////////////////////////

import java.util.ArrayList;

import java.util.List;

/\*\*

\* An abstract base class providing some functionality of the BinaryTree

\* Interface.

\* @author Steven Glasford, Goodrick, Tamassia, Goldwasser

\* Data Structures & Algorithms 6th Edition

\* @version 3-5-2019

\* @param <E> A generic parameter

\*/

public abstract class AbstractBinaryTree<E> extends AbstractTree<E>

implements BinaryTree<E> {

/\*\*

\* Returns the Position of p s sibling (or null if no sibling exists).

\* @param p The position of the other sibling for a node.

\* @return The position of the sibling.

\*/

@Override

public Position<E> sibling(Position<E> p){

Position<E> parent = parent(p);

//p must be the root

if(parent == null) return null;

//p is a left child

if (p == left(parent))

//(right child might be null)

return right(parent);

//p is a right child

else

//(left child might be null)

return left(parent);

}

/\*\*

\* Returns the number of children of Position p.

\* @param p The node you are testing for.

\* @return The number of children for a particular node, in int.

\*/

@Override

public int numChildren(Position<E> p) {

int count = 0;

if (left(p) != null)

count++;

if (right(p) != null)

count++;

return count;

}

/\*\*

\* Returns an iterable collection of the Positions representing p children.

\* @param p The position you want to mess around with.

\* @return The iterable collection of the Positions representing

\* p children.

\*/

@Override

public Iterable<Position<E>> children(Position<E> p) {

//max capacity of 2

List<Position<E>> snapshot = new ArrayList<>(2);

if (left(p) != null)

snapshot.add(left(p));

if (right(p) != null)

snapshot.add(right(p));

return snapshot;//by progressive

}

/\*\*

\* Adds positions of the subtree rooted at Position p to the

\* given snapshot.

\* @param p The position to begin with.

\* @param snapshot The lists of positions in which the thing is located.

\*/

private void inorderSubtree(Position<E> p, List<Position<E>> snapshot){

if (left(p) != null)

inorderSubtree(left(p),snapshot);

snapshot.add(p);

if (right(p) != null)

inorderSubtree(right(p),snapshot);

}

/\*\*

\* Returns an iterable collection of positions of the tree, reported

\* in inorder.

\* @return an iterable collection of positions of the tree, reported

\* in inorder.

\*/

public Iterable<Position<E>> inorder() {

List<Position<E>> snapshot = new ArrayList<>();

if (!isEmpty())

//fill the snapshot recursively

inorderSubtree(root(),snapshot);

return snapshot;

}

/\*\*

\* Overrides positions to make inorder the default for binary trees.

\* @return an inorder operator.

\*/

@Override

public Iterable<Position<E>> positions(){

return inorder();

}

}

/////////////////////////////////////////////////////////////////////////////

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

/\*\*

\* An abstract base class providing some functionality of the Tree interface.

\* @author Steven Glasford, Goodrick, Tamassia, Goldwasser

\* Data Structures & Algorithms 6th Edition.

\* @param <E> A generic parameter

\*/

public abstract class AbstractTree<E> implements Tree<E> {

/\*\*

\* Determine if the node that is being tested is an internal

\* component.

\* @param p The node you want to determine if it is an internal

\* component.

\* @return true or false depending on whether or not the test node

\* is an internal component.

\*/

@Override

public boolean isInternal(Position<E> p) {return numChildren(p) > 0;}

/\*\*

\* Determine if a node is an external component, if it is a leaf.

\* @param p The node you want to test.

\* @return True or false dependent on the conditions.

\*/

@Override

public boolean isExternal(Position<E> p) {return numChildren(p) == 0;}

/\*\*

\* Determine if a node is a Root (like my car insurance).

\* @param p The node you want to test.

\* @return True or false, depending on if the node is a root.

\*/

@Override

public boolean isRoot(Position<E> p) {return p == root();}

/\*\*

\* Determine if a tree is empty.

\* @return True or false, depending on if the tree is empty.

\*/

@Override

public boolean isEmpty() {return size() == 0;}

/\*\*

\* Returns the number of levels separating Position p from the root.

\* @param p The node you want to test at.

\* @return The number of levels separating Position p from the root.

\*/

public int depth(Position<E> p) {

if (isRoot(p))

return 0;

else

return 1 + depth(parent(p));

}

/\*\*

\* Returns the height of the tree.

\* Works, but has quadratic worst-case time.

\* @return the height of the tree.

\*/

private int heightBad(){

int h = 0;

for (Position<E> p : positions())

//only consider leaf positions

if (isExternal(p))

h = Math.max(h, depth(p));

return h;

}

/\*\*

\* Returns the height of the subtree rooted at Position p.

\* @param p The node you are testing from.

\* @return the height of the tree

\*/

public int height(Position<E> p){

//base case if p is external

int h = 0;

for (Position<E> c : children(p))

h = Math.max(h, 1 + height(c));

return h;

}

/\*\*

\* This class adapts the iteration produced by positions() to return

\* elements.

\*/

private class ElementIterator implements Iterator<E> {

Iterator<Position<E>> posIterator = positions().iterator();

@Override

public boolean hasNext() {return posIterator.hasNext();}

//return element

@Override

public E next() {return posIterator.next().getElement();}

@Override

public void remove() {posIterator.remove();}

}

/\*\*

\* Returns an iterator of the elements stored in the tree.

\* @return an iterator of the elements stored in the tree.

\*/

@Override

public Iterator<E> iterator() {return new ElementIterator();}

/\*\*

\* defining the preorder as the default traversal algorithm for the

\* public positions method of an abstract tree.

\* @return

\*/

@Override

public Iterable<Position<E>> positions() {return preorder();}

/\*\*

\* Adds positions of the subtree rooted at Position p to the given

\* snapshot.

\* @param p A position to be investigated

\* @param snapshot by Progressive.

\*/

private void preorderSubtree(Position<E> p, List<Position<E>> snapshot){

//for preorder, we add position p before exploring subtrees

snapshot.add(p);

for(Position<E> c : children(p))

preorderSubtree(c,snapshot);

}

/\*\*

\* Returns an iterable collection of positions of the tree,

\* reported in preorder.

\* @return an iterable collection of positions of the tree,

\* reported in preorder.

\*/

public Iterable<Position<E>> preorder(){

List<Position<E>> snapshot = new ArrayList<>();

if (isEmpty())

//fill the snapshot recursively

preorderSubtree(root(), snapshot);

return snapshot;

}

/\*\*

\* Adds positions of the subtree rooted at Position p to the

\* given snapshot

\* @param p The position of the subtree

\* @param snapshot by progressive

\*/

private void postorderSubtree(Position<E> p, List<Position<E>> snapshot){

for (Position<E> c : children(p))

postorderSubtree(c,snapshot);

//for postorder, we add position p after exploring subtrees

snapshot.add(p);

}

/\*\*

\* Returns an iterable collection of positions of the tree,

\* reported in postorder.

\* @return an iterable collection of positions of the tree,

\* reported in postorder.

\*/

public Iterable<Position<E>> postorder(){

List<Position<E>> snapshot = new ArrayList<>();

if (!isEmpty())

//fill the snapshot recursively

postorderSubtree(root(), snapshot);

return snapshot;

}

/\*\*

\* Returns an iterable collection of positions of the tree in

\* breadth-first order.

\* @return an iterable collection of positions of the tree in

\* breadth-first order.

\*/

public Iterable<Position<E>> breadthfirst(){

List<Position<E>> snapshot = new ArrayList<>();

if (!isEmpty()) {

Queue<Position<E>> fringe = new LinkedQueue<>();

//start with the root

fringe.enqueue(root());

while (!fringe.isEmpty()) {

//remove from the front of the queue

Position<E> p = fringe.dequeue();

//report this position

snapshot.add(p);

for (Position<E> c : children(p))

//add children to the back of the queue

fringe.enqueue(c);

}

}

return snapshot;

}

}

/////////////////////////////////////////////////////////////////////////////

/\*\*

\* An interface for a binary tree, in which each node has at most two children.

\* @author Steven Glasford, Goodrick, Tamassia, Goldwasser

\* Data Structures & Algorithms 6th Edition

\* @version 3-5-2019

\* @param <E> A generic parameter.

\*/

public interface BinaryTree<E> extends Tree<E> {

/\*\*

\* Returns the Position of p s left child (or null if no child exists).

\* @param p The position you want to find the left child for.

\* @return The position of the left child from an input position.

\* @throws IllegalArgumentException If the node doesnt exist.

\*/

Position<E> left(Position<E> p) throws IllegalArgumentException;

/\*\*

\* Returns the Position of p s right child (or null if no child exists).

\* @param p The position of the parent node.

\* @return The position of the right child.

\* @throws IllegalArgumentException If the input position doesnt exist.

\*/

Position<E> right(Position<E> p) throws IllegalArgumentException;

/\*\*

\* Returns the Position of p s sibling (or null if no sibling exists).

\* @param p The node you want to find its sibling for.

\* @return The position of the other sibling.

\* @throws IllegalArgumentException If the input position doesnt exist.

\*/

Position<E> sibling(Position<E> p) throws IllegalArgumentException;

}

/////////////////////////////////////////////////////////////////////////////

import java.util.Iterator;

/\*\*

\* Concrete implementation of a binary tree using a node-based, linked

\* structure.

\* @author Steven Glasford, Goodrick, Tamassia, Goldwasser

\* Data Structures & Algorithms 6th Edition.

\* @version 3-5-2019

\* @param <E> A generic parameter

\*/

public class LinkedBinaryTree<E> extends AbstractBinaryTree<E> {

@Override

public Iterator<E> iterator() {

throw new UnsupportedOperationException("Not supported yet.");

//To change body of generated methods, choose Tools | Templates.

}

@Override

public Iterable<Position<E>> positions() {

throw new UnsupportedOperationException("Not supported yet.");

//To change body of generated methods, choose Tools | Templates.

}

//------------------nested Node class------------------//

/\*\*

\* A commonality of all nodes in the tree, smells like potatoes.

\* @param <E> A generic input parameter.

\*/

protected static class Node<E> implements Position<E> {

//an element stored at this node.

private E element;

//a reference to the parent node (if any).

private Node<E> parent;

//a reference to the left child (if any).

private Node<E> left;

//a reference to the right child (if any).

private Node<E> right;

/\*\*

\* Constructs a node with the given element and neighbors.

\* @param e The element you want to add.

\* @param above The parent of the new child.

\* @param leftChild The left child of the new parent.

\* @param rightChild The right child of the new parent.

\*/

public Node(E e, Node<E> above, Node<E> leftChild,

Node<E> rightChild) {

element = e;

parent = above;

left = leftChild;

right = rightChild;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* accessor methods \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

/\*\*

\* Gets element in the node out of protection.

\* @return The data within the tree.

\*/

@Override

public E getElement() {return element;}

/\*\*

\* Gets the parent out of protections, assuming they are due on

\* alimony payments.

\* @return The parent to the child, null if the root.

\*/

public Node<E> getParent() {return parent;}

/\*\*

\* Get the left child out of protection.

\* @return Return the node of the left child.

\*/

public Node<E> getLeft() {return left;}

/\*\*

\* Get the right child out of protection.

\* @return Return the node of the left child.

\*/

public Node<E> getRight() {return right;}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* update methods \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

/\*\*

\* Set the element from outside of protection.

\* @param e The element you want to add to the node.

\*/

public void setElement(E e) {element = e;}

/\*\*

\* Set the nodes parent, like adoption.

\* @param parentNode The parent node.

\*/

public void setParent(Node<E> parentNode) { parent = parentNode;}

/\*\*

\* Set the left Child of the node.

\* @param leftChild The left child you want to set.

\*/

public void setLeft(Node<E> leftChild) { left = leftChild; }

/\*\*

\* Set the right Child of the node.

\* @param rightChild The right child that you want to set.

\*/

public void setRight(Node<E> rightChild) { right = rightChild; }

} /////////////////////// end of nested Node class /////////////////////

/\*\*

\* Factory function to create a new node storing element e.

\* @param e The element you want to create a node for.

\* @param parent The parent of the new node you just made.

\* @param left The first, left, child of the new node.

\* @param right The second, right, child of the new node.

\* @return The new node.

\*/

protected Node<E> createNode(E e, Node<E> parent, Node<E> left,

Node<E> right){

return new Node<>(e,parent,left,right);

}

///////////////////LinkedBinaryTree instance variables///////////////////

//root of the tree, protected like dirt.

protected Node<E> root = null;

//number of nodes in the tree

private int size = 0;

/\*\*

\* Constructor, constructs an empty binary tree

\*/

public LinkedBinaryTree(){}

///////////////////////nonpublic utility////////////////////////////

/\*\*

\* Validates the position and returns it as a node.

\* @param p The position you want to create.

\* @return The node from the position.

\* @throws IllegalArgumentException If the position doesnt exist.

\*/

protected Node<E> validate(Position<E> p) throws IllegalArgumentException {

if (!(p instanceof Node))

throw new IllegalArgumentException("Not valid position type");

//safe cast

Node<E> node = (Node<E>) p;

//Our convention for defunct node

if (node.getParent() == node)

throw new IllegalArgumentException("p is no longer in the tree");

return node;

}

//////Accesor methods (not already implemented in AbstractBinaryTree)//////

/\*\*

\* Returns the number of nodes in the tree.

\* @return An integer of the size of the tree.

\*/

@Override

public int size() {

return size;

}

/\*\*

\* Returns the root Position of the tree (or null if tree is empty).

\* @return The position of the root.

\*/

@Override

public Position<E> root(){

return root;

}

/\*\*

\* Returns the Positions of p s parent (or null if p is root)

\* @param p The position you are testing from.

\* @return The position of the parent.

\* @throws IllegalArgumentException if the position doesnt exist.

\*/

@Override

public Position<E> parent(Position<E> p) throws IllegalArgumentException {

Node<E> node = validate(p);

return node.getParent();

}

/\*\*

\* Returns the Position of p s left child (or null if no child exists).

\* @param p The node you are trying to find the child for.

\* @return The position of the left nodes left child.

\* @throws IllegalArgumentException if the position doesnt exist.

\*/

@Override

public Position<E> left(Position<E> p) throws IllegalArgumentException {

Node<E> node = validate(p);

return node.getLeft();

}

/\*\*

\* Returns the Position of p s right child (or null if no child exists)

\* @param p The position you are trying to find the right child for.

\* @return The position of the right child.

\* @throws IllegalArgumentException if the input position doesnt exist.

\*/

@Override

public Position<E> right(Position<E> p) throws IllegalArgumentException {

Node<E> node = validate(p);

return node.getRight();

}

///////////////update methods supported by this class///////////////////

/\*\*

\* Places element e at the root of an empty tree and returns

\* its new position.

\* @param e The element you want to be the root of the tree.

\* @return The new position of the root of the tree.

\* @throws IllegalStateException if the tree is not empty.

\*/

public Position<E> addRoot(E e) throws IllegalStateException {

if (!isEmpty()) throw new IllegalStateException("Tree is not empty");

root = createNode(e, null, null, null);

size = 1;

return root;

}

/\*\*

\* Creates a new left child of Position p storing element e;

\* returns its Position.

\* @param p The parent of the new child.

\* @param e The element you want to add to the new child.

\* @return The position of the newly created left child.

\* @throws IllegalArgumentException if the input position already

\* has a child.

\*/

public Position<E> addLeft(Position<E> p, E e)

throws IllegalArgumentException{

Node<E> parent = validate(p);

if (parent.getLeft() != null)

throw new IllegalArgumentException("p already has a right child");

Node<E> child = createNode(e,parent,null,null);

parent.setLeft(child);

size++;

return child;

}

/\*\*

\* Creates a new right child of Position p storing element e;

\* returns its Position.

\* @param p The parent of the new child.

\* @param e The element you want to add to the new child.

\* @return The position of the new right child.

\* @throws IllegalArgumentException if the position already has a

\* right child.

\*/

public Position<E> addRight(Position<E> p, E e)

throws IllegalArgumentException{

Node<E> parent = validate(p);

if (parent.getRight() != null)

throw new IllegalArgumentException("p already has a right child");

Node<E> child = createNode(e, parent, null, null);

parent.setRight(child);

size++;

return child;

}

/\*\*

\* Replaces the element at Position p with e and returns the

\* replaced element

\* @param p The position in which you are changing the element data for.

\* @param e The element you want to set the data for.

\* @return The element you set for the particular node.

\* @throws IllegalArgumentException if the position and node do not exist.

\*/

public E set(Position<E> p, E e) throws IllegalArgumentException {

Node<E> node = validate(p);

E temp = node.getElement();

node.setElement(e);

return temp;

}

/\*\*

\* Attaches trees t1 and t2 as left and right subtrees of external p.

\* @param p The new parent of the two trees.

\* @param t1 The left part of the tree.

\* @param t2 The right part of the tree.

\* @throws IllegalArgumentException if one node doesnt exist.

\*/

public void attach(Position<E> p, LinkedBinaryTree<E> t1,

LinkedBinaryTree<E> t2) throws IllegalArgumentException {

Node<E> node = validate(p);

if (isInternal(p)) throw new IllegalArgumentException("p must"

+ " be a leaf");

size += t1.size() + t2.size();

//attach t1 as left subtree of node

if (!t1.isEmpty()){

t1.root.setParent(node);

node.setLeft(t1.root);

t1.root = null;

t1.size = 0;

}

//attach t2 as right subtree of node

if (!t2.isEmpty()){

t2.root.setParent(node);

node.setRight(t2.root);

t2.root = null;

t2.size = 0;

}

}

/\*\*

\* Removes the node at Position p and replaces it with its child,

\* if any.

\* @param p The position of the node you want to remove.

\* @return The element that once was stored in the element you

\* just removed.

\* @throws IllegalArgumentException if the element doesn't exist.

\*/

public E remove(Position<E> p) throws IllegalArgumentException {

Node<E> node = validate(p);

if (numChildren(p) == 2)

throw new IllegalArgumentException("p has two children");

Node<E> child = (node.getLeft() != null ? node.getLeft():

node.getRight());

if (child != null)

//child s grandparent becomes its parent

child.setParent(node.getParent());

if (node == root)

//child becomes root

root = child;

else{

Node<E> parent = node.getParent();

if (node == parent.getLeft())

parent.setLeft(child);

else

parent.setRight(child);

}

size--;

E temp = node.getElement();

//help garbage collection

node.setElement(null);

node.setLeft(null);

node.setRight(null);

//our convention for defunct node

node.setParent(node);

return temp;

}

/\*\* Prints parenthesized representation of subtree of T rooted at p.

\* @param <E>

\* @param T

\* @param p

\*/

public static <E> void parenthesize(Tree<E> T, Position<E> p) {

System.out.print(p.getElement( ));

if (T.isInternal(p)) {

boolean firstTime = true;

for (Position<E> c : T.children(p)) {

// determine proper punctuation

System.out.print( (firstTime ? " (" : ", ") );

// any future passes will get comma

firstTime = false;

// recur on child

parenthesize(T, c);

}

System.out.print(")");

}

}

/////////////////////end of LinkedBinaryTree class////////////////////////

}

/////////////////////////////////////////////////////////////////////////////

/\*\*

\* Realization of a FIFO queue as an implementation of a SinglyLinkedSet.

\*

\* @author Michael T. Goodrich

\* @author Roberto Tamassia

\* @author Michael H. Goldwater

\* @author Steven Glasford

\* @version 2-21-2019

\* @param <E>

\*/

public class LinkedQueue<E> implements Queue<E> {

//an empty list

private final SinglyLinkedList<E> list = new SinglyLinkedList<>();

//new queue relies on the initially empty list

public LinkedQueue() {}

@Override

public int size() {return list.size();}

@Override

public boolean isEmpty() {return list.isEmpty();}

@Override

public void enqueue(E element) {list.addLast(element);}

@Override

public E first() {return list.first();}

@Override

public E dequeue() {return list.removeFirst();}

}

/////////////////////////////////////////////////////////////////////////////

/\*\*

\* @author Michael T. Goodrich

\* @author Roberto Tamassia

\* @author Michael H. Goldwater

\* @author Steven Glasford

\* @version 2-21-2019

\* @param <E>

\*/

public class LinkedStack<E> implements Stack<E> {

//an empty list

private final SinglyLinkedList<E> list = new SinglyLinkedList<>();

//new stack relies on the initially empty list

public LinkedStack() {}

@Override

public int size() {return list.size();}

@Override

public boolean isEmpty() {return list.isEmpty();}

@Override

public void push(E element) { list.addFirst(element); }

@Override

public E top() { return list.first(); }

@Override

public E pop() { return list.removeFirst(); }

}

/////////////////////////////////////////////////////////////////////////////

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragement 7.7

\*/

public interface Position<E> {

/\*\*

\* Returns the element stored at this position.

\*

\* @return the stored element

\* @thorws IllegalStateExceptoin if position no longer valid

\*/

E getElement( ) throws IllegalStateException;

}

/////////////////////////////////////////////////////////////////////////////

/\*\*

\* @author Michael T. Goodrich

\* @author Roberto Tamassia

\* @author Michael H. Goldwater

\* @author Steven Glasford

\* @version 2-21-2019

\* @param <E>

\*/

public interface Queue<E> {

/\*\*

\* Returns the number of elements in the queue

\* @return

\*/

int size();

/\*\*

\* Tests whether the queue is empty

\* @return

\*/

boolean isEmpty();

/\*\*

\* Inserts an element at the rear of the queue

\* @param e

\* @todo modify so that this is required to throw a queue Full Exception

\* if called on a full queue

\*/

void enqueue(E e);

/\*\*

\* returns, but does not remove, the first element of the queue

\* (null if empty).

\* @return

\*/

E first();

/\*\*

\* Removes and returns the first element of the queue (null if empty)

\* @return

\*/

E dequeue();

}

/////////////////////////////////////////////////////////////////////////////

/\*\*

\*

\* SinglyLinkedList Class

\* Code Fragments 3.14, 3.15

\* from

\* Data Structures & Algorithms, 6th edition

\* by Michael T. Goodrich, Roberto Tamassia & Michael H. Goldwasser

\* Wiley 2014

\* Transcribed by

\* @author Steven Glasford

\* @version January 31, 2019

\* @param <E> a generic placeholder name

\*/

public class SinglyLinkedList<E> {

/\*\*

\*

\* @param <E> a generic placeholder name

\*

\* A subclass creating the Node

\*/

private static class Node<E>{

//reference to the element stored at this node

private final E element;

//reference to the subsequent node in the list

private Node<E> next;

public Node(E e, Node<E> n){

element = e;

next = n;

}

/\*\*

\*

\* @return Return the current element

\*/

public E getElement(){return element;}

/\*\*

\*

\* @return return the address of the next item in the linked list

\*/

public Node<E> getNext() {return next;}

/\*\*

\*

\* @param n the next item in the list

\*/

public void setNext(Node<E> n) {next = n;}

}

//head node of the list (or null if empty)

private Node<E> head = null;

//last node of the list (or null if empty)

private Node<E> tail = null;

//number of nodes in the list

private int count = 0;

/\*\*

\* constructs an initially empty list

\*/

public SinglyLinkedList(){}

//access methods

/\*\*

\*

\* @return Return the size of the linked list

\*/

public int size() {return count;}

/\*\*

\*

\* @return Determine if the linked list is empty

\*/

public boolean isEmpty() {return count == 0;}

/\*\*

\*

\* @return return the first element in the list

\*

\* returns (but does not remove) the first element

\*/

public E first(){

if (isEmpty()) return null;

return head.getElement();

}

/\*\*

\*

\* @return the last element in the linked list

\*

\* returns (but does not remove the last element

\*/

public E last(){

if (isEmpty()) return null;

return tail.getElement();

}

//update methods

/\*\*

\*

\* @param e A generic element

\*

\* adds element e to the front of the list

\*/

public void addFirst(E e){

//create and link a new node

head = new Node<>(e, head);

//special case: new node becomes tail also

if (count == 0)

tail = head;

count++;

}

/\*\*

\*

\* @param e A generic item

\*

\* adds element e to the end of the list

\*/

public void addLast(E e) {

//node will eventually be the tail

Node<E> newest = new Node<>(e,null);

//special case: previously empty list

if (isEmpty())

head = newest;

else

tail.setNext(newest);

tail = newest;

count++;

}

/\*\*

\*

\* @return return the item that was removed

\*

\* removes and returns the first element

\*/

public E removeFirst(){

//nothing to remove

if (isEmpty()) return null;

E answer = head.getElement();

//will become null if list had only one node

head = head.getNext();

count--;

//special case as list is now empty

if(count == 0)

tail = null;

return answer;

}

}

/////////////////////////////////////////////////////////////////////////////

/\*\*

\* A collection of objects that are inserted and removed according to the

\* last-in first-out principle; although similar in purpose, this

\* interface differs from "java.util.Stack"

\*

\* @author Michael T. Goodrich

\* @author Roberto Tamassia

\* @author Michael H. Goldwater

\*

\* @version 2-21-2019s

\*/

public interface Stack<E> {

/\*\*

\* Returns the number of elements in the stack

\* @return number of elements in the stack

\*/

int size();

/\*\*

\* Tests whether the stack is empty.

\* @return true if the stack is empty, false otherwise.

\*/

boolean isEmpty();

/\*\*

\* Inserts an element at the top of the stack

\* @param e the element to be inserted

\* @todo modify so this method is required to trow a Stack Full

\* exception if called on a full stack

\*/

void push(E e);

/\*\*

\* Returns, but does not remove , the element at the top of the stack

\* @return to element in the stack (or null if empty)

\*/

E top();

/\*\*

\* Removes and returns the top element from the stack.

\* @return element removed (or null if empty)

\*/

E pop();

}

/////////////////////////////////////////////////////////////////////////////

import java.util.Iterator;

/\*\*

\* An interface for a tree where nodes can have an arbitrary number of children

\* @author Steven Glasford, Goodrick, Tamassia, Goldwasser

\* Data Structures & Algorithms 6th Edition

\* @version 3-5-2019

\* @param <E> A generic parameter

\*/

public interface Tree<E> extends Iterable<E> {

/\*\*

\* Make the root of the tree.

\* @return The root of the tree.

\*/

Position<E> root();

/\*\*

\* Make a parent of in the tree.

\* @param p The leaf you want to make into a parent.

\* @return A new parent node

\* @throws IllegalArgumentException If the node doesnt exist.

\*/

Position<E> parent(Position<E> p) throws IllegalArgumentException;

/\*\*

\* Make a child, without any of the fun sex positions.

\* @param p

\* @return

\* @throws IllegalArgumentException if the node doesnt exist.

\*/

Iterable<Position<E>> children(Position<E> p)

throws IllegalArgumentException;

/\*\*

\* Determine how many children a catholic has.

\* @param p The catholic you want to determine the number of children for.

\* @return The number of children raped by the priest (all of them).

\* @throws IllegalArgumentException if the node doesn't exist.

\*/

int numChildren(Position<E> p) throws IllegalArgumentException;

/\*\*

\* Determine if the node is an internal node within the tree.

\* @param p The node you want to test.

\* @return Whether or not the node is an internal component.

\* @throws IllegalArgumentException if the node doesnt exist.

\*/

boolean isInternal(Position<E> p) throws IllegalArgumentException;

/\*\*

\* Determine if the node is an external, whether it is a leaf.

\* @param p The node that you want to test.

\* @return Whether or not the node is a leaf.

\* @throws IllegalArgumentException If the node doesnt exist.

\*/

boolean isExternal(Position<E> p) throws IllegalArgumentException;

/\*\*

\* Determine if a node is a root node.

\* @param p The node you want to test.

\* @return Whether or not the node is a root.

\* @throws IllegalArgumentException If the node doesnt exist.

\*/

boolean isRoot(Position<E> p) throws IllegalArgumentException;

/\*\*

\* Determine the size of the tree.

\* @return An integer of the number of nodes in the tree.

\*/

int size();

/\*\*

\* Determine if the tree is empty.

\* @return Whether or not the tree is empty.

\*/

boolean isEmpty();

/\*\*

\* An iterator of the tree for easy passage through the tree.

\* @return an iterator.

\*/

@Override

Iterator<E> iterator();

/\*\*

\* The position of the tree, usually this is a node, but can be a root,

\* like ginseng or ginger.

\* @return The iterable thing.

\*/

Iterable<Position<E>> positions();

}

/////////////////////////////////////////////////////////////////////////////

3 \* -5

4.5 m 3.6 / 5.2

4.5 - 3.6 / 5.2

9.1 + 6.3 \* 5.0

( 4 - 3 ) / 5

4 + ( 7 / 2 )

[ 4 + 7 ] \* { 8 - 11 }

4 + 7 8 - 11

( ( [ 3 + 1 ] \* 3 ) / ( ( 9 - 5 ) ) - ( ( 3 \* ( 7 - 4 ) ) + 6 ) )

( ( 3 + 1 ) \* 3 ) / ( ( 9 - 5 ) ) - ( ( 3 \* ( 7 - 4 ) ) + 6 ) )

3 + 1 \* 3 / 9 - 5 - 3 \* 7 - 4 + 6

42

8 \* 24 / ( 4 + 3

3 + 4 –

-4 \* -4

/////////////////////////////////////////////////////////////////////////////

run:

Expression: 3 \* -5

Post Fix: 3 -5 \*

Pre Order: \* -5 3

In Order: -5 \* 3

Post Order: -5 3 \*

Eulers Tour: \* (-5, 3)

Evaluated: -15.0

Expression: 4.5 m 3.6 / 5.2

java.lang.RuntimeException: Expression does not have correct amount operators or operands

Expression: 4.5 - 3.6 / 5.2

Post Fix: 4.5 3.6 5.2 / -

Pre Order: - / 5.2 3.6 4.5

In Order: 5.2 / 3.6 - 4.5

Post Order: 5.2 3.6 / 4.5 -

Eulers Tour: - (/ (5.2, 3.6), 4.5)

Evaluated: 3.5

Expression: 9.1 + 6.3 \* 5.0

Post Fix: 9.1 6.3 5.0 \* +

Pre Order: + \* 5.0 6.3 9.1

In Order: 5.0 \* 6.3 + 9.1

Post Order: 5.0 6.3 \* 9.1 +

Eulers Tour: + (\* (5.0, 6.3), 9.1)

Evaluated: 3.0

Expression: ( 4 - 3 ) / 5

Post Fix: 4 3 - 5 /

Pre Order: / 5 - 3 4

In Order: 5 / 3 - 4

Post Order: 5 3 4 - /

Eulers Tour: / (5, - (3, 4))

Evaluated: 3.4

Expression: 4 + ( 7 / 2 )

Post Fix: 4 7 2 / +

Pre Order: + / 2 7 4

In Order: 2 / 7 + 4

Post Order: 2 7 / 4 +

Eulers Tour: + (/ (2, 7), 4)

Evaluated: 7.5

Expression: [ 4 + 7 ] \* { 8 - 11 }

Post Fix: 4 7 + 8 11 - \*

Pre Order: \* - 11 8 + 7 4

In Order: 11 - 8 \* 7 + 4

Post Order: 11 8 - 7 4 + \*

Eulers Tour: \* (- (11, 8), + (7, 4))

Evaluated: -129.0

Expression: 4 + 7 8 - 11

java.lang.RuntimeException: Expression does not have correct amount operators or operands

Expression: ( ( [ 3 + 1 ] \* 3 ) / ( ( 9 - 5 ) ) - ( ( 3 \* ( 7 - 4 ) ) + 6 ) )

Post Fix: 3 1 + 3 \* 9 5 - / 3 7 4 - \* 6 + -

Pre Order: - + 6 \* - 4 7 3 / - 5 9 \* 3 + 1 3

In Order: 6 + 4 - 7 \* 3 - 5 - 9 / 3 \* 1 + 3

Post Order: 6 4 7 - 3 \* + 5 9 - 3 1 3 + \* / -

Eulers Tour: - (+ (6, \* (- (4, 7), 3)), / (- (5, 9), \* (3, + (1, 3))))

Evaluated: -38.5

Expression: ( ( 3 + 1 ) \* 3 ) / ( ( 9 - 5 ) ) - ( ( 3 \* ( 7 - 4 ) ) + 6 ) )

Post Fix: 3 1 + 3 \* 9 5 - / 3 7 4 - \* 6 + -

Pre Order: - + 6 \* - 4 7 3 / - 5 9 \* 3 + 1 3

In Order: 6 + 4 - 7 \* 3 - 5 - 9 / 3 \* 1 + 3

Post Order: 6 4 7 - 3 \* + 5 9 - 3 1 3 + \* / -

Eulers Tour: - (+ (6, \* (- (4, 7), 3)), / (- (5, 9), \* (3, + (1, 3))))

Evaluated: -38.5

Expression: 3 + 1 \* 3 / 9 - 5 - 3 \* 7 - 4 + 6

Post Fix: 3 1 3 9 / \* 5 3 7 \* 4 6 + - - - +

Pre Order: + - - - + 6 4 \* 7 3 5 \* / 9 3 1 3

In Order: 6 + 4 - 7 \* 3 - 5 - 9 / 3 \* 1 + 3

Post Order: 6 4 + 7 3 \* - 5 - 9 3 / 1 \* - 3 +

Eulers Tour: + (- (- (- (+ (6, 4), \* (7, 3)), 5), \* (/ (9, 3), 1)), 3)

Evaluated: -36.333333333333336

Expression: 42

Post Fix: 42

Pre Order: 42

In Order: 42

Post Order: 42

Eulers Tour: 42

Evaluated: 42.0

Expression: 8 \* 24 / ( 4 + 3

java.lang.RuntimeException: Brackets incomplete

Expression: 3 + 4 –

java.lang.RuntimeException: Expression does not have correct amount operators or operands

Expression: -4 \* -4

Post Fix: -4 -4 \*

Pre Order: \* -4 -4

In Order: -4 \* -4

Post Order: -4 -4 \*

Eulers Tour: \* (-4, -4)

Evaluated: 16.0

BUILD SUCCESSFUL (total time: 26 seconds)