Dylan T Carlson

1161653

Lab109

11/1/18

**Client:**

/\*\*

\* This Client requests a data file, reads it in, and evaluates

\* the expressions that are on each line in the file. It uses

\* the shunting yard algorithm to do this.

\*

\* @author Dylan Carlson

\*/

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Iterator;

import java.util.Scanner;

// Left for easy copy and pasting for future usage.

// C:\Users\dylca\Desktop\Lab109\_Test.txt

public class Client {

public static void main(String[] args) throws FileNotFoundException{

Scanner scan3 = new Scanner(System.in);

System.out.println("Please input a file path: ");

String path = scan3.nextLine();

File data = new File(path);

Scanner mainScan = new Scanner(data);

while(mainScan.hasNext()){

System.out.println("----New Expression----");

String expression = mainScan.nextLine();

Queue<String> postFixed = inFixedToPostFixed( stringToQueue(expression) );

Queue<String> postFixed2 = inFixedToPostFixed( stringToQueue(expression) );

Queue<String> tokens = stringToQueue(expression);

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

System.out.println("This expression equals: " );

double answer = evaluate(postFixed2);

if(!(Math.abs(answer + 1) < 0.1) && (postFixed2 != null) ){

System.out.print(answer);

LinkedBinaryTree<String> binaryTree = binaryExpressionTree(postFixed);

System.out.println("\nExpression: ");

while(tokens.size() != 0){

System.out.print(tokens.dequeue());

}

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

System.out.print(" Pre Order: ");

Iterator<Position<String>> preOrderIterator = binaryTree.preorder().iterator();

while(preOrderIterator.hasNext()){

System.out.print( preOrderIterator.next().getElement());

System.out.print(" ");

}

System.out.print("\n\n In Order: ");

Iterator<Position<String>> inOrderIterator = binaryTree.inorder().iterator();

while(inOrderIterator.hasNext()){

System.out.print( inOrderIterator.next().getElement());

System.out.print(" ");

}

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

System.out.print(" Post Order: ");

Iterator<Position<String>> postOrderIterator = binaryTree.postorder().iterator();

while(postOrderIterator.hasNext()){

System.out.print( postOrderIterator.next().getElement());

System.out.print(" ");

}

System.out.println("\n\nEuler Tour: ");

binaryTree.eulerTourBinary(binaryTree, binaryTree.root);

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

}

}

}

/\*\*

\* stringToQueue is passed an expression from a data file, and

\* converts it into a queue that is used later.

\*

\* @param express

\* @return

\* @throws FileNotFoundException

\*/

public static Queue<String> stringToQueue( String express ) throws FileNotFoundException{

Scanner scan;

scan = new Scanner( express );

Queue<String> expression = new LinkedQueue<>();

while( scan.hasNext() ){

expression.enqueue( scan.next() );

}

return expression;

}

/\*\* Shunting yard algorithm \*/

/\*\*

\* inFixedToPostFixed takes a queue of strings as an input

\* and returns another queue of strings, but in post fixed notation.

\*

\* @param input

\* @return

\*/

public static Queue<String> inFixedToPostFixed(Queue<String> input) {

Stack<String> operators = new LinkedStack<>();

Queue<String> postFixed = new LinkedQueue<>();

try{

while(input.first() != null){

switch(input.first()){

case "\*" :

while(operators.size() != 0 && (operators.top().equals("/") || operators.top().equals("\*")) )

postFixed.enqueue( operators.pop() );

operators.push(input.dequeue());

break;

case "/" :

while(operators.size() != 0 && operators.top().equals("/"))

postFixed.enqueue( operators.pop() );

operators.push(input.dequeue());

break;

case "+" :

while(operators.size() != 0 && (operators.top().equals("\*") || operators.top().equals("/") || operators.top().equals("-")) )

postFixed.enqueue( operators.pop() );

operators.push(input.dequeue());

break;

case "-" :

while(operators.size() != 0 && (operators.top().equals("\*") || operators.top().equals("/") || operators.top().equals("-")) )

postFixed.enqueue( operators.pop() );

operators.push(input.dequeue());

break;

case "(" :

operators.push(input.dequeue());

break;

case "[" :

operators.push(input.dequeue());

break;

case "{" :

operators.push(input.dequeue());

break;

case ")" :

boolean T = true;

while(T == true){

switch(operators.top()){

case "(" :

operators.pop();

input.dequeue();

T = false;

break;

case "{" :

case "[" :

throw new IllegalArgumentException();

default :

postFixed.enqueue(operators.pop());

}

}

break;

case "]" :

T = true;

while(T == true){

switch(operators.top()){

case "[" :

operators.pop();

input.dequeue();

T = false;

break;

case "(" :

case "{" :

throw new IllegalArgumentException();

default :

postFixed.enqueue(operators.pop());

}

}

break;

case "}" :

T = true;

while(T == true){

switch(operators.top()){

case "{" :

operators.pop();

input.dequeue();

T = false;

break;

case "(" :

case "[" :

throw new IllegalArgumentException();

default :

postFixed.enqueue(operators.pop());

}

}

break;

default :

postFixed.enqueue(input.dequeue());

}

}

//Popping the rest of the operators into the queue

while(operators.top() != null){

postFixed.enqueue(operators.pop());

}

}

catch (IllegalArgumentException | NullPointerException iae){

return null;

}

return postFixed;

}

/\*\*

\* evaluate is passed the post fixed notation of an expression of one

\* of the lines of the data file. It returns the result of the expression.

\*

\* @param input

\* @return

\* @throws IllegalArgumentException

\*/

public static double evaluate(Queue<String> input) throws IllegalArgumentException{

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

double result = 0;

double op1;

double op2;

try{

while(input.first() != null){

switch(input.first()){

case "/" :

op1 = integers.pop();

op2 = integers.pop();

result = op2 / op1;

integers.push(result);

input.dequeue();

break;

case "\*" :

op1 = integers.pop();

op2 = integers.pop();

result = op2 \* op1;

integers.push(result);

input.dequeue();

break;

case "-" :

op1 = integers.pop();

op2 = integers.pop();

result = op2 - op1;

integers.push(result);

input.dequeue();

break;

case "+" :

op1 = integers.pop();

op2 = integers.pop();

result = op2 + op1;

integers.push(result);

input.dequeue();

break;

default:

double newEntry = Double.parseDouble(input.dequeue());

integers.push(newEntry);

}

}

if(integers.size() == 1)

result = integers.pop();

else

throw new IllegalArgumentException();

}

catch (NullPointerException npe) {

System.out.println("Invalid Expression (Null pointer)");

return -1;

}

catch (IllegalArgumentException iae) {

System.out.println("Invalid Expression");

return -1;

}

return result;

}

/\*\*

\* binaryExpressionTree is passed a queue of strings. It makes a

\* binary expression tree for the given expression from post fixed

\* notation. The tree is used for the multiple traversals of the tree.

\*

\* @param input

\* @return

\*/

public static LinkedBinaryTree<String> binaryExpressionTree( Queue<String> input) {

Stack<LinkedBinaryTree<String>> stackOfTrees = new LinkedStack<>();

LinkedBinaryTree<String> result = new LinkedBinaryTree<>();

while(input.first() != null){

LinkedBinaryTree<String> sub1 = new LinkedBinaryTree<>();

LinkedBinaryTree<String> sub2 = new LinkedBinaryTree<>();

LinkedBinaryTree<String> sub3 = new LinkedBinaryTree<>();

switch(input.first()){

case "/" :

sub1 = stackOfTrees.pop();

sub2 = stackOfTrees.pop();

sub3.addRoot(input.dequeue());

sub3.attach(sub3.root(), sub2, sub1);

stackOfTrees.push(sub3);

break;

case "\*" :

sub1 = stackOfTrees.pop();

sub2 = stackOfTrees.pop();

sub3.addRoot(input.dequeue());

sub3.attach(sub3.root(), sub2, sub1);

stackOfTrees.push(sub3);

break;

case "-" :

sub1 = stackOfTrees.pop();

sub2 = stackOfTrees.pop();

sub3.addRoot(input.dequeue());

sub3.attach(sub3.root(), sub2, sub1);

stackOfTrees.push(sub3);

break;

case "+" :

sub1 = stackOfTrees.pop();

sub2 = stackOfTrees.pop();

sub3.addRoot(input.dequeue());

sub3.attach(sub3.root(), sub2, sub1);

stackOfTrees.push(sub3);

break;

default:

sub1.addRoot(input.dequeue());

stackOfTrees.push(sub1);

}

}

result = stackOfTrees.pop();

return result;

}

}

**Abstract Binary Tree:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 8.7, 8.22

\* eulerTourBinary created by user with help from

\* code fragment 8.29

\*/

import java.util.ArrayList;

import java.util.List;

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

public abstract class AbstractBinaryTree<E> extends AbstractTree<E> implements BinaryTree<E> {

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

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

Position <E> parent = parent(p);

if(parent == null) return null; // p must be the root

if(p == left(parent)) // p is a left child

return right(parent); // (right child might be null)

else // p is a right child

return left(parent); // (left child might be null)

}

/\*\* Returns the number of children of Position p. \*/

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's children. \*/

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

List<Position<E>> snapshot = new ArrayList<>(2); // max capacity of 2

if(left(p) != null)

snapshot.add(left(p));

if(right(p) != null)

snapshot.add(right(p));

return snapshot;

}

/\*\* Code fragment 8.22. \*/

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

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 order. \*/

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

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

if(!isEmpty())

inorderSubtree(root(), snapshot); // fill the snapshot recursively

return snapshot;

}

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

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

return inorder();

}

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

if(isInternal(p))

System.out.print("(");

if(left(p) != null)

eulerTourBinary(T, left(p));

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

if(right(p) != null)

eulerTourBinary(T, right(p));

if(isInternal(p))

System.out.print(")");

}

}

**Abstract Tree:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 8.2, 8.3, 8.5, 8.16, 8.17, 8.18, 8.19, 8.20,

\* 8.21, 8.23, 8.24, 8.25, 8.26, 8.27

\*/

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

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

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

private static <E> String spaces(int i) {

String space = "";

for(int j = 0; j < i; j++){

space += " ";

}

return space;

}

/\*\* Methods from 8.2. \*/

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

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

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

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

/\*\* Method depth from 8.3. \*/

/\*\* Returns 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));

}

/\*\* Method height (better one) from 8.5. \*/

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

public int height(Position<E> p) {

int h = 0;

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

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

return h;

}

/\*\* Method from 8.16. \*/

//-------- nested ElementIterator class --------

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

private class ElementIterator implements Iterator<E> {

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

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

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

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

}

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

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

/\*\* Code fragment 8.17. \*/

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

/\*\* Code fragment 8.18. \*/

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

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

snapshot.add(p); // for preorder, we add postion p before exploring subtrees

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

preorderSubtree(c, snapshot);

}

/\*\* Code fragment 8.19. \*/

/\*\* Returns an iterable collection of positions of the tree, reported in preorder. \*/

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

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

if(!isEmpty())

preorderSubtree(root(), snapshot); // fill the snapshot recursively

return snapshot;

}

/\*\* Code fragment 8.20. \*/

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

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

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

postorderSubtree(c, snapshot);

snapshot.add(p); // for postorder, we add position p after exploring subtrees

}

/\*\* Returns an interable collection of positions of the tree, reported in postorder. \*/

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

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

if(!isEmpty())

postorderSubtree(root(), snapshot); // fill the snapshot recursively

return snapshot;

}

/\*\* Code fragment 8.21. \*/

/\*\* Returns 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<>();

fringe.enqueue(root()); // start with the root

while(!fringe.isEmpty()) {

Position<E> p = fringe.dequeue(); // remove from front of queue

snapshot.add(p); // report this position

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

fringe.enqueue(c); // add children to back of queue

}

}

return snapshot;

}

/\*\* Code fragment 8.23. \*/

/\*\* Prints preorder representation of subtree of T rooted at p having depth d. \*/

public static <E> void printPreorderIndent(Tree<E> T, Position<E> p, int d) {

System.out.println(spaces(2\*d) + p.getElement()); // indent based on d

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

printPreorderIndent(T, c, d+1);

}

/\*\* Code fragment 8.24. \*/

/\*\* Prints labeled representation of subtree of T rooted at p having depth d. \*/

public static <E> void printPreorderLabeled(Tree<E> T, Position<E> p, ArrayList<Integer> path){

int d = path.size(); // depth equals the length of the path

System.out.println(spaces(2\*d)); // print indentation, then label

for(int j = 0; j < d; j++) System.out.println(path.get(j) + (j == d-1 ? " " : "."));

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

path.add(1); // add path entry for first child

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

printPreorderLabeled(T, c, path);

path.set(d, 1 + path.get(d)); // increment last entry of path

}

path.remove(d); // restore path to its incoming state

}

/\*\* Not Needed But kept for future reference. \*/

// /\*\* Code fragment 8.25. \*/

// /\*\* Returns total disk space for subtree of T rooted at p. \*/

// public static int diskSpace(Tree<Integer> T, Position<Integer> p) {

// int subtotal = p.getElement(); // we assume element represents space usage

// for(Position<Integer> c: T.children(p))

// subtotal += diskSpace(T, c);

// return subtotal;

// }

/\*\* Code fragment 8.26. \*/

/\*\* Prints parenthesized representation of subtree of T rooted at 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)) {

System.out.print( (firstTime ? " (" : ", ") ); // determine proper punctuation

firstTime = false; // any future passes will get comma

parenthesize(T, c); // recur on child

}

System.out.print(")");

}

}

/\*\* Not Needed But kept for future reference. \*/

//

// /\*\* Code fragment 8.27. \*/

// public static <E> int layout(BinaryTree<E> T, Position<E> p, int d, int x) {

// if(T.left(p) != null)

// x = layout(T, T.left(p), d+1, x); // resulting x will be increased

// p.getElement().setX(x++); // post-increment x

// p.getElement().setY(d);

// if(T.right(p) != null)

// x = layout(T, T.right(p), d+1, x); // resulting x will be increased

// return x;

// }

}

**Binary Tree Interface:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 8.6

\*/

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

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

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

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

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

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

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

}

**Linked Binary Tree:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragments 8.8, 8.9, 8.10, 8.11

\*/

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

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

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

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

private E element; // an element stored at this node

private Node<E> parent; // a reference to the parent node (if any)

private Node<E> left; // a reference to the left child (if any)

private Node<E> right; // a reference to the right child (if any)

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

public Node(E e, Node<E> above, Node<E> leftChild, Node<E> rightChild) {

element = e;

parent = above;

left = leftChild;

right = rightChild;

}

// accessor methods

public E getElement() { return element; }

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

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

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

// update methods

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

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

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

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

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

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

protected Node<E> createNode(E e, Node<E> parent, Node<E> left, Node<E> right) {

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

}

// LinkedBinaryTree instance variables

protected Node<E> root = null; // root of the tree

private int size = 0; // number of nodes in the tree

// constructor

public LinkedBinaryTree(){} // constructs an empty binary tree

// nonpublic utility

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

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

if(!(p instanceof Node))

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

Node<E> node = (Node<E>) p; // safe cast

if(node.getParent() == node) // our convention for defunct node

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

return node;

}

// accessor methods (not already implemented in AbstractBinaryTree)

/\*\* Returns the number of nodes in the tree. \*/

public int size() {

return size;

}

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

public Position<E> root(){

return root;

}

/\*\* Returns the Position of p's parent (or null if p is root). \*/

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). \*/

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). \*/

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. \*/

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. \*/

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 left 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. \*/

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. \*/

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 subtree of external p. \*/

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();

if(!t1.isEmpty()) {

t1.root.setParent(node); // attach t1 as left subtree of node

node.setLeft(t1.root);

t1.root = null;

t1.size = 0;

}

if (!t2.isEmpty()) {

t2.root.setParent(node); // attach t2 as right subtree of 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. \*/

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.setParent(node.getParent()); // child's grandparent becomes its parent

if (node == root)

root = child; // child becomes root

else {

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

if (node == parent.getLeft())

parent.setLeft(child);

else

parent.setRight(child);

}

size--;

E temp = node.getElement();

node.setElement(null); // help garbage collection

node.setLeft(null);

node.setRight(null);

node.setParent(node); // our convention for defunct node

return temp;

}

} //-------- end of LinkedBinaryTree class --------

**LinkedQueue:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 6.11

\*

\* An implementation of the LinkedQueue class

\*/

/\*\* Realization of a FIFO queue as an adaptation of a SinglyLinkedList. \*/

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

private SinglyLinkedList<E> list = new SinglyLinkedList<>(); // an empty list

public LinkedQueue() {} // new queue relies on the iitially empty list

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

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

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

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

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

}

**LinkedStack**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 6.4

\*

\* An implementation of the LinkedStack class

\*/

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

private SinglyLinkedList<E> list = new SinglyLinkedList<>(); //an empty list

public LinkedStack() {} //new stack relies on the initially empty list

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

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

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

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

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

}

**Position Interface:**

/\*\*

\* 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;

}

**Queue Interface:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 6.9

\*

\* An implementation of the Queue interface

\*/

public interface Queue<E> {

/\*\* Returns the number of elements in the queue. \*/

int size();

/\*\* Tests whether the queue is empty. \*/

boolean isEmpty();

/\*\* Inserts an element at the rear of the queue. \*/

void enqueue( E e);

/\*\* Returns, but does not remove, the first element of the queue (null if empty). \*/

E first();

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

E dequeue();

}

**SinglyLinkedList:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 3.14 and 3.15

\*

\* An implementation of a SinglyLinkedList class

\*/

public class SinglyLinkedList<E> {

//------Nested Node class------

private static class Node<E>{

private E element;

private Node<E> next;

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

element = e;

next = n;

}

public E getElement(){ return element; }

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

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

}

//------End of Nested Node class------

//Instance variables of SinglyLinkedList

private Node<E> head = null;

private Node<E> tail = null;

private int size = 0;

public SinglyLinkedList(){}

//access methods

public int size(){ return size; }

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

public E first(){

if( isEmpty() ) return null;

return head.getElement();

}

public E last(){

if( isEmpty() ) return null;

return tail.getElement();

}

//Update methods

public void addFirst(E e){

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

if( size == 0)

tail = head;

size++;

}

public void addLast(E e){

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

if ( isEmpty() )

head = newest;

else

tail.setNext(newest);

tail = newest;

size++;

}

public E removeFirst(){

if ( isEmpty() ) return null;

E answer = head.getElement();

head = head.getNext();

size--;

if( size == 0 )

tail = null;

return answer;

}

}

**Stack Interface:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 6.1

\*

\* An implementation of the generic Stack interface

\*/

/\*\*

\* 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.until.Stack.

\*

\* @author Michael T. Goodrick

\* @author Roberto Tamassia

\* @author Michael H. Goldwasser

\*/

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

\*/

void push(E e);

/\*\*

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

\* @return top 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();

}

**Tree Interface:**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragment 8.1

\*/

import java.util.Iterator;

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

Position<E> root();

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

Iterable<Position<E>> children(Position<E> p) throws IllegalArgumentException;

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

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

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

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

int size();

boolean isEmpty();

Iterator<E> iterator();

Iterable<Position<E>> positions();

}

**OutPut:**

run:

Please input a file path:

C:\Users\dylca\Desktop\Lab109\_Test.txt

----New Expression----

This expression equals:

50.794117647058826

Expression:

(24\*2)-7/(9+25)+3

Pre Order: + - \* 24 2 / 7 + 9 25 3

In Order: 24 \* 2 - 7 / 9 + 25 + 3

Post Order: 24 2 \* 7 9 25 + / - 3 +

Euler Tour:

(((24\*2)-(7/(9+25)))+3)

----New Expression----

This expression equals:

27.006999999999998

Expression:

12+(300.14/20)

Pre Order: + 12 / 300.14 20

In Order: 12 + 300.14 / 20

Post Order: 12 300.14 20 / +

Euler Tour:

(12+(300.14/20))

----New Expression----

This expression equals:

31.125

Expression:

10+3\*8-3/4/6\*7-2

Pre Order: + 10 - - \* 3 8 \* / / 3 4 6 7 2

In Order: 10 + 3 \* 8 - 3 / 4 / 6 \* 7 - 2

Post Order: 10 3 8 \* 3 4 / 6 / 7 \* - 2 - +

Euler Tour:

(10+(((3\*8)-(((3/4)/6)\*7))-2))

----New Expression----

This expression equals:

-4.0

Expression:

2-2+2-2-2-2+2-2

Pre Order: + - 2 2 + - - - 2 2 2 2 - 2 2

In Order: 2 - 2 + 2 - 2 - 2 - 2 + 2 - 2

Post Order: 2 2 - 2 2 - 2 - 2 - 2 2 - + +

Euler Tour:

((2-2)+((((2-2)-2)-2)+(2-2)))

BUILD SUCCESSFUL (total time: 9 seconds)

**Screen Shots:**



