import java.io.File;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.util.Scanner;

/\*\*

\* @author Samuel Swedberg

\* @version 11/4/22

\*

\* A client that demonstrates a shunting yard algorithm with arithmetic expressions.

\*/

public class Client {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

String directory = "", answer = "";

boolean tryAgain = true;

while(tryAgain)

{

boolean inValidInput = true;

//Loops until valid input is found

while(inValidInput)

{

System.out.println("Enter starting entire directory with filename: ");

directory = sc.nextLine();

try {

if(!new File(directory).exists())

throw new FileNotFoundException("Error: FileNotFoundException.");

inValidInput = false;

// FileNotFoundException

} catch (FileNotFoundException fnfe) {

System.out.println(fnfe.getMessage() + " Press a key to continue or N to leave: ");

answer = sc.nextLine();

// Asks user if they want to continue

if(answer.equalsIgnoreCase("N"))

tryAgain = false;

break;

}

}

// Runs only if valid input is found

if(!inValidInput)

{

File file = new File(directory);

LinkedQueue lineQueue = new LinkedQueue();

LinkedQueue charQueue = new LinkedQueue();

try

{

if(!new File(directory).exists())

throw new FileNotFoundException();

Scanner scanLine = new Scanner( file );

/\*

// Scans line

while( scanLine.hasNext() )

{

String stringLine = scanLine.nextLine();

lineQueue.enqueue(stringLine);

System.out.println(stringLine);

}\*/

while( scanLine.hasNext() )

{

String stringWord = scanLine.nextLine();

lineQueue.enqueue(stringWord);

System.out.println("Infix Expression: " + stringWord);

// Splits word into chars

String stringChar[] = stringWord.split(" ");

for(String character : stringChar)

{

charQueue.enqueue(character);

}

postFixEvaluate(charQueue);

}

scanLine.close();

} catch ( FileNotFoundException fnfe )

{

System.out.println("Unable to find file.");

}

catch ( IOException ioe )

{

ioe.printStackTrace();

}

// start

// end

System.out.println("Press a key to continue or N to leave: ");

answer = sc.nextLine();

// Asks user if they want to continue

if(answer.equalsIgnoreCase("N"))

tryAgain = false;

}

}

}

/\*

postFixEvaluate method

@param queue

@return void

\*/

public static void postFixEvaluate(LinkedQueue queue)

{

boolean inValidExpression = false;

// Shunting yard data structures

LinkedQueue rQ = queue; // Right Queue

LinkedQueue lQ = new LinkedQueue(); // Left Queue

LinkedStack bS = new LinkedStack(); // Bottom Stack

// This for loop removes parenthesis and will move all values to the lQ

int size1 = rQ.size();

for(int i=0; i<size1; i++) // Runs through rQ

{

//System.out.println("Loop: " + i);

if(isOperator(rQ.first())) // Is an operator

{

//System.out.println("Is Operator");

if(isParRight(rQ.first())) // Checks if rQ is )

{

//System.out.println("Is )");

int bsSize = bS.size();

for(int j=0; j<bsSize; j++) // Runs through bS

{

//System.out.println(") Loop: " + j);

if(isParLeft(bS.top())) // Checks if top of stack is (

{

//System.out.println(" Is (");

//System.out.println("Popped: " + bS.top());

bS.pop(); // Pops (

break; // Breaks out of for loop (this only needs to find the first left parentheses)

}

else

{

//System.out.println("Pushing: " + bS.top());

lQ.enqueue(bS.pop()); // Pops top of stack to lQ

}

if(bS.isEmpty())

{

inValidExpression = true;

break;

}

}

rQ.dequeue(); // Removes )

}

else

{

//System.out.println("Pushing: " + rQ.first());

bS.push(rQ.dequeue());

}

}

else

{

//System.out.println("Enqueueing lQ: " + rQ.first());

lQ.enqueue(rQ.dequeue()); // Enqueues if a number

}

if(rQ.isEmpty()) // If rQ is empty then moves all of the remaining bS to lQ

{

while(!(bS.isEmpty())) // Runs until bS is empty

{

lQ.enqueue(bS.pop()); // Pops bS to lQ

}

}

}

rQ = lQ; // Moves lQ to rQ;

lQ = null; // Deletes lQ;

LinkedQueue copyRq = new LinkedQueue();

int copySize = rQ.size();

for(int c=0; c<copySize; c++)

{

Object temp = rQ.dequeue();

copyRq.enqueue(temp);

rQ.enqueue(temp);

}

String printPostFix = "";

for (int x=0; x<rQ.size(); x++)

{

String temp = ( String ) rQ.dequeue();

printPostFix += temp + "";

rQ.enqueue(temp);

}

System.out.println("Postfix Expression: " + printPostFix);

int size2 = rQ.size();

// This for loop will evaluate

for(int k=0; k<size2; k++)

{

if(!(inValidExpression))

{

if(!(rQ.isEmpty()))

{

if(isOperator(rQ.first())) // Checks if is an operator

{

//System.out.println("Is Operator");

if(bS.top() == null)

{

inValidExpression = true;

break;

}

//System.out.println("lV: " + bS.top());

String s1 = String.valueOf(bS.pop());

double rV = Double.parseDouble(s1); // Converts object to double

if(bS.top() == null)

{

inValidExpression = true;

break;

}

//System.out.println("rV: " + bS.top());

String s2 = String.valueOf(bS.pop());

double lV = Double.parseDouble(s2); // Converts object to double

if(rV == 0)

{

inValidExpression = true;

break;

}

else if(rQ.first().equals("(")|| rQ.first().equals(")"))

{

inValidExpression = true;

break;

}

else

{

bS.push(doMath(lV, rV, rQ.first())); // Does calculation and pushes back down into bS

//System.out.println("doMath: " + doMath(lV, rV, rQ.first()));

rQ.dequeue(); // Removes operator

}

}

else

{

//System.out.println("Pushing: " + rQ.first());

bS.push(rQ.dequeue()); // Pushes number from rQ

}

}

}

}

if(inValidExpression)

{

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

}

else

{

System.out.println("Valid expression.");

System.out.println("Expression evaluation: " + bS.top()); // Prints final value

postFixToTree(copyRq); // Converts postfix expression to a tree

}

Scanner sc = new Scanner(System.in);

System.out.println("Press a key to continue");

String rando = sc.nextLine();

}

/\*

postFixToTree method

@param queue

@return void

\*/

public static void postFixToTree(LinkedQueue queue)

{

LinkedBinaryTree treeFinal = new LinkedBinaryTree();

LinkedStack stackTree = new LinkedStack();

while(!(queue.isEmpty()))

{

//System.out.println("Queue Size: " + queue.size());

if(!(isOperator(queue.first()))) // Creates one node tree and pushes to expression stack

{

//System.out.println("Is Operand: " + queue.first());

LinkedBinaryTree tree = new LinkedBinaryTree();

tree.addRoot(queue.dequeue());

stackTree.push(tree);

}

else // Creates new tree with operator as root, and pops tree off expression stack for left and right children. Pushes new tree onto stack

{

//System.out.println("Is Operator: " + queue.first());

LinkedBinaryTree tree = new LinkedBinaryTree();

tree.addRoot(queue.dequeue());

LinkedBinaryTree lC = (LinkedBinaryTree) stackTree.pop();

LinkedBinaryTree rC = (LinkedBinaryTree) stackTree.pop();

tree.attach(tree.root(), lC, rC);

stackTree.push(tree);

}

}

treeFinal = (LinkedBinaryTree) stackTree.top();

System.out.println("Preorder: " + preOrder(treeFinal)); // Prints preorder traversal

System.out.println("Inorder: " + inOrder(treeFinal)); // Prints inorder traversal

System.out.println("Postorder: " + postOrder(treeFinal)); // Prints postorder traversal

System.out.println("Breadthfirst: " + breadthFirst(treeFinal)); // Prints breadthfirst

System.out.print("Eulertour: "); // Prints eulertour

eulerTour(treeFinal, treeFinal.root());

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

}

/\*

isOperator method

@param o

@return true/false

\*/

public static boolean isOperator(Object o)

{

String s = ( String ) o;

switch(s) {

case "+":

return true;

case "-":

return true;

case "\*":

return true;

case "/":

return true;

case "(":

return true;

case ")":

return true;

case "{":

return true;

case "}":

return true;

case "[":

return true;

case "]":

return true;

default:

return false;

}

}

/\*

isParRight method

@param o

@return true/false

\*/

public static boolean isParRight(Object o)

{

String s = ( String ) o;

switch(s) {

case ")":

return true;

case "}":

return true;

case "]":

return true;

default:

return false;

}

}

/\*

isParLeft method

@param o

@return true/false

\*/

public static boolean isParLeft(Object o)

{

String s = ( String ) o;

switch(s) {

case "(":

return true;

case "{":

return true;

case "[":

return true;

default:

return false;

}

}

/\*

doMath method

@param lV, rV, o

@return calcuation

\*/

public static double doMath(double lV, double rV, Object o)

{

//System.out.println("lV: " + lV + ", rV: " + rV + ", O: " + o);

String s = ( String ) o;

switch(s) {

case "+":

return lV + rV;

case "-":

return lV - rV;

case "\*":

return lV \* rV;

case "/":

return lV / rV;

}

return 0;

}

/\*

preOrder method

@param t

@return returnString, returns preOrder t in a string

\*/

public static <E> String preOrder(LinkedBinaryTree<E> t)

{

String returnString = "";

for(Position<E> p : t.preorder())

{

returnString = returnString + p.getElement() + " ";

}

return returnString;

}

/\*

inOrder method

@param t

@return returnString, returns inOrder t in a string

\*/

public static <E> String inOrder(LinkedBinaryTree<E> t)

{

String returnString = "";

for(Position<E> p : t.inorder())

{

returnString = returnString + p.getElement() + " ";

}

return returnString;

}

/\*

postOrder method

@param t

@return returnString, returns postOrder t in a string

\*/

public static <E> String postOrder(LinkedBinaryTree<E> t)

{

String returnString = "";

for(Position<E> p : t.postorder())

{

returnString = returnString + p.getElement() + " ";

}

return returnString;

}

/\*

breadthFirst method

@param t

@return returnString, returns breadthFirst t in a string

\*/

public static <E> String breadthFirst(LinkedBinaryTree<E> t)

{

String returnString = "";

for(Position<E> p : t.breadthfirst())

{

returnString = returnString + p.getElement() + " ";

}

return returnString;

}

/\*

eulerTour method

@param t

Prints out left and right children using the euler tour

\*/

public static <E> void eulerTour(LinkedBinaryTree<E> t, Position<E> p)

{

System.out.print("(");

if(t.left(p) != null)

{

eulerTour(t,t.left(p));

}

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

if(t.right(p) != null)

{

eulerTour(t,t.right(p));

}

System.out.print(")");

}

}

/\*\*

\* AbstractBinaryTree Class

\* Code Fragments from Chapter 8

\* from

\* Data Structures & Algorithms, 6th edition

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

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 11/4/22

\*/

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;

}

/\*\* 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 inorder. \*/

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( );

}

}

/\*\*

\* Abstract Tree Class

\* Code Fragments from Chapter 8

\* from

\* Data Structures & Algorithms, 6th edition

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

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 11/4/22

\*/

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> {

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; }

/\*\* 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));

}

/\*\* Returns the height of the tree. \*/

private int heightBad( ) { // works, but quadratic worst-case time

int h = 0;

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

if (isExternal(p)) // only consider leaf positions

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

return h;

}

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

public int height(Position<E> p) {

int h = 0; // base case if p is external

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

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

return h;

}

/\*\* 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 position p before exploring subtrees

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

preorderSubtree(c, snapshot);

}

/\*\* 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;

}

/\*\* 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 iterable 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;

}

/\*\* 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 the queue

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

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

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

}

}

return snapshot;

}

//---------------- 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( ); }

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

}

/\*\*

\* BinaryTree Class

\* Code Fragments from Chapter 8

\* from

\* Data Structures & Algorithms, 6th edition

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

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 10/25/22

\*/

/\*\* 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). \*/

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

}

/\*\*

\* LinkedBinaryTree Class

\* Code Fragments from Chapter 8

\* from

\* Data Structures & Algorithms, 6th edition

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

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 11/4/22

\*/

/\*\* 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 subtrees 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( )) { // attach t1 as left subtree of node

t1.root.setParent(node);

node.setLeft(t1.root);

t1.root = null;

t1.size = 0;

}

if (!t2.isEmpty( )) { // attach t2 as right subtree of node

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. \*/

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 Class

\* Code Fragments 6.11

\* from

\* Data Structures & Algorithms, 6th edition

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

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 11/4/22

\*/

/\*\* 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 initially 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 Class

\* Code Fragments 6.4

\* from

\* Data Structures & Algorithms, 6th edition

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

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 11/4/22

\*/

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 Class

\* Code Fragments from Chapter 8

\* from

\* Data Structures & Algorithms, 6th edition

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

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 11/4/22

\*/

public interface Position<E> {

/\*\*

\* Returns the element stored at this position.

\*

\* @return the stored element

\* @throws IllegalStateException if position no longer valid

\*/

E getElement( ) throws IllegalStateException;

}

/\*\*

\* Queue Class

\* Code Fragments 6.9

\* from

\* Data Structures & Algorithms, 6th edition

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

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 11/4/22

\*/

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 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 Samuel Swedberg

\* @version 11/4/22

\*/

public class SinglyLinkedList<E> {

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

private static class Node<E> {

private E element; // reference to the element stored at this node

private Node<E> next; // reference to the subsequent node in the list

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; }

}

// instance variables of the SinglyLinkedList

private Node<E> head = null; // head node of the list (or null if empty)

private Node<E> tail = null; // last node of the list (or null if empty)

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

public SinglyLinkedList() {} // constructs an initially empty list

// access methods

public int size( ) { return size; }

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

public E first( ) { // returns (but does not remove) the first element

if (isEmpty( )) return null;

return head.getElement( );

}

public E last( ) { // returns (but does not remove) the last element

if (isEmpty( )) return null;

return tail.getElement( );

}

// update methods

public void addFirst(E e) { // adds element e to the front of the list

head = new Node<>(e, head); // create and link a new node

if (size == 0)

tail = head; // special case: new node becomes tail also

size++;

}

public void addLast(E e) { // adds element e to the end of the list

Node<E> newest = new Node<>(e, null); // node will eventually be the tail

if (isEmpty( ))

head = newest; // special case: previously empty list

else

tail.setNext(newest); // new node after existing tail

tail = newest; // new node becomes the tail

size++;

}

public E removeFirst( ) { // removes and returns the first element

if (isEmpty( )) return null; // nothing to remove

E answer = head.getElement( );

head = head.getNext( ); // will become null if list had only one node

size--;

if (size == 0)

tail = null; // special case as list is now empty

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.

\*

\* Stack Class

\* Code Fragments 6.1

\* from

\* Data Structures & Algorithms, 6th edition

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

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 11/4/22

\*/

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 Class

\* Code Fragments from Chapter 8

\* from

\* Data Structures & Algorithms, 6th edition

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

\* Wiley 2014

\* Transcribed by

\* @author Samuel Swedberg

\* @version 11/4/22

\*/

import java.util.Iterator;

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

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( );

}

Text

Description automatically generated

Text

Description automatically generated