import java.util.Iterator;

import java.util.List;

/\*\*

\*

\* @author Samuel Swedberg

\* @version 10/25/22

\*

\* A client class that demonstrates ways to travel along a LinkedBinaryTree

\*/

public class Client {

public static void main(String[] args) {

LinkedBinaryTree root0 = new LinkedBinaryTree();

LinkedBinaryTree root1 = new LinkedBinaryTree();

LinkedBinaryTree root2 = new LinkedBinaryTree();

LinkedBinaryTree root5 = new LinkedBinaryTree();

LinkedBinaryTree root12 = new LinkedBinaryTree();

LinkedBinaryTree root11 = new LinkedBinaryTree();

LinkedBinaryTree root6 = new LinkedBinaryTree();

root0.addRoot("+"); // 0

root1.addRoot("\*"); // 1

root1.addLeft(root1.root(), "9"); // 3

root1.addRight(root1.root(), "8"); // 4

root2.addRoot("\*"); // 2

root5.addRoot("/"); // 5

root6.addRoot("-2"); // 6

root11.addRoot("17"); // 11

root12.addRoot("+"); // 12

root12.addLeft(root12.root(), "14"); // 25

root12.addRight(root12.root(), "23"); // 26

root5.attach(root5.root(), root11, root12); // Attaches 11 and 12

root2.attach(root2.root(), root5, root6); // Attaches 5 and 6

root0.attach(root0.root(), root1, root2); // Attaches 1 & 2

String expression = "( 9 \* 8 ) + ( 17 / ( 14 + 23 ) \* -2 )"; // Prints expression as a string

System.out.println("Expression: " + expression);

System.out.println("Height: " + root0.height(root0.root())); // Prints height

System.out.println("Preorder: " + preOrder(root0)); // Prints preorder traversal

System.out.println("Inorder: " + inOrder(root0)); // Prints inorder traversal

System.out.println("Postorder: " + postOrder(root0)); // Prints postorder traversal

System.out.println("Breadthfirst: " + breadthFirst(root0)); // Prints breadthfirst

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

eulerTour(root0, root0.root());

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

}

/\*

preOrder method

@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

@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

@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

@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

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 10/25/22

\*/

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 10/25/22

\*/

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;

}

/\*\*

\* 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 10/25/22

\*/

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;

}

/\*\*

\* 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 10/25/22

\*/

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( );

}

run:

Expression: ( 9 \* 8 ) + ( 17 / ( 14 + 23 ) \* -2 )

Height: 4

Preorder: + \* 9 8 \* / 17 + 14 23 -2

Inorder: 9 \* 8 + 17 / 14 + 23 \* -2

Postorder: 9 8 \* 17 14 23 + / -2 \* +

Breadthfirst: + \* \* 9 8 / -2 17 + 14 23

Eulertour: (((9)\*(8))+(((17)/((14)+(23)))\*(-2)))

BUILD SUCCESSFUL (total time: 0 seconds)