Dylan T Carlson

1161653

Lab108

10/19/18

**Client Class:**

/\*\*

\* In this client I create the correct binary tree for the

\* arithmetic expression defined. It is then printed out in several ways.

\*

\* @author Dylan Carlson

\*/

import java.util.Iterator;

public class Client {

public static void main(String[] args) {

LinkedBinaryTree lbt = new LinkedBinaryTree(); // main tree

LinkedBinaryTree temp1 = new LinkedBinaryTree();

LinkedBinaryTree temp2 = new LinkedBinaryTree();

LinkedBinaryTree temp3 = new LinkedBinaryTree();

LinkedBinaryTree temp4 = new LinkedBinaryTree();

LinkedBinaryTree temp5 = new LinkedBinaryTree();

LinkedBinaryTree temp6 = new LinkedBinaryTree();

LinkedBinaryTree temp7 = new LinkedBinaryTree();

LinkedBinaryTree temp8 = new LinkedBinaryTree();

LinkedBinaryTree temp9 = new LinkedBinaryTree();

LinkedBinaryTree temp10 = new LinkedBinaryTree();

LinkedBinaryTree sub1 = new LinkedBinaryTree();

LinkedBinaryTree sub2 = new LinkedBinaryTree();

LinkedBinaryTree sub3 = new LinkedBinaryTree();

LinkedBinaryTree sub4 = new LinkedBinaryTree();

LinkedBinaryTree sub5 = new LinkedBinaryTree();

LinkedBinaryTree sub6 = new LinkedBinaryTree();

LinkedBinaryTree sub7 = new LinkedBinaryTree();

LinkedBinaryTree sub8 = new LinkedBinaryTree();

temp1.addRoot("2");

temp2.addRoot("1");

sub1.addRoot("-");

sub1.attach(sub1.root, temp1, temp2);

temp3.addRoot("5");

temp4.addRoot("2");

sub2.addRoot("+");

sub2.attach(sub2.root, temp3, temp4);

sub3.addRoot("\*");

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

sub4.addRoot("+");

temp5.addRoot("2");

temp6.addRoot("9");

sub4.attach(sub4.root, temp5, temp6);

sub5.addRoot("/");

sub5.attach(sub5.root, sub3, sub4);

sub6.addRoot("-");

temp7.addRoot("7");

temp8.addRoot("2");

sub6.attach(sub6.root, temp7, temp8);

sub7.addRoot("-");

temp9.addRoot("1");

sub7.attach(sub7.root, sub6, temp9);

sub8.addRoot("+");

sub8.attach(sub8.root, sub5, sub7);

temp10.addRoot("8");

lbt.addRoot("\*");

lbt.attach(lbt.root, sub8, temp10);

lbt.eulerTourBinary(lbt, lbt.root);

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

Iterator<Position<String>> inOrderIterator = lbt.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 = lbt.postorder().iterator();

while(postOrderIterator.hasNext()){

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

System.out.print(" ");

}

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

System.out.print(" Breadth first Order: ");

Iterator<Position<String>> breadthFirstIterator = lbt.breadthfirst().iterator();

while(breadthFirstIterator.hasNext()){

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

System.out.print(" ");

}

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

System.out.println(" preOrder Indent: ");

lbt.printPreorderIndent(lbt, lbt.root, 1);

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

System.out.println(" parenthesized repersentation: ");

lbt.parenthesize(lbt, lbt.root);

}

}

**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();

}

**BinaryTree 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) . \*/

}

**AbstractTree Class:**

/\*\*

\* 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;

// }

}

**AbstractBinaryTree Class:**

/\*\*

\* 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(")");

}

}

**LinkedBinaryTree Class:**

/\*\*

\* 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 --------

**Output:**

run:

(((((5+2)\*(2-1))/(2+9))+((7-2)-1))\*8)

In Order: 5 + 2 \* 2 - 1 / 2 + 9 + 7 - 2 - 1 \* 8

Post Order: 5 2 + 2 1 - \* 2 9 + / 7 2 - 1 - + 8 \*

Breadth first Order: \* + 8 / - \* + - 1 + - 2 9 7 2 5 2 2 1

preOrder Indent:

\*

+

/

\*

+

5

2

-

2

1

+

2

9

-

-

7

2

1

8

parenthesized repersentation:

\* (+ (/ (\* (+ (5, 2), - (2, 1)), + (2, 9)), - (- (7, 2), 1)), 8)BUILD SUCCESSFUL (total time: 0 seconds)