**ClientLab108**

import java.util.ArrayList;

/\*\*

\*

\* @author Marie Larson

\* @version 3/21/18

\*/

public class ClientLab108 {

/\*\*

\* @param args the command line arguments

\*/

public static void main(String[] args) {

System.out.println("Begin Lab108");

System.out.println("Creating the following equation in a "

+ "linkedBinaryTree: (((5+2)\*(2-1)/(2+9)+((7-2)-1))\*8) ");

//creating bottom branch

LinkedBinaryTree<String> LBT1 = new LinkedBinaryTree<>();

LBT1.addRoot("+");

LBT1.addLeft(LBT1.root, "5");

LBT1.addRight(LBT1.root, "2");

LinkedBinaryTree<String> LBT2 = new LinkedBinaryTree<>();

LBT2.addRoot("-");

LBT2.addLeft(LBT2.root, "2");

LBT2.addRight(LBT2.root, "1");

LinkedBinaryTree<String> LBT3 = new LinkedBinaryTree<>();

LBT3.addRoot("\*");

LBT3.attach(LBT3.root, LBT1, LBT2);

LinkedBinaryTree<String> LBT4 = new LinkedBinaryTree<>();

LBT4.addRoot("+");

LBT4.addLeft(LBT4.root, "2");

LBT4.addRight(LBT4.root, "9");

LinkedBinaryTree<String> LBT5 = new LinkedBinaryTree<>();

LBT5.addRoot("/");

LBT5.attach(LBT5.root, LBT3, LBT4);

LinkedBinaryTree<String> LBT6 = new LinkedBinaryTree<>();

LBT6.addRoot("-");

LBT6.addLeft(LBT6.root, "7");

LBT6.addRight(LBT6.root, "2");

LinkedBinaryTree<String> LBT8 = new LinkedBinaryTree<>();

LBT8.addRoot("1");

LinkedBinaryTree<String> LBT7 = new LinkedBinaryTree<>();

LBT7.addRoot("-");

LBT7.attach(LBT7.root, LBT7, LBT8);

LinkedBinaryTree<String> LBT9 = new LinkedBinaryTree<>();

LBT9.addRoot("+");

LBT9.attach(LBT9.root, LBT5, LBT7);

LinkedBinaryTree<String> LBT11 = new LinkedBinaryTree<>();

LBT11.addRoot("8");

LinkedBinaryTree<String> LBT10 = new LinkedBinaryTree<>();

LBT10.addRoot("\*");

LBT10.attach(LBT10.root, LBT9, LBT11);

System.out.println("\n\nA The preoder traversal of the tree");

for(Position<String> p : LBT10.preOrder())

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

System.out.println("\n\n The inOrder traversal of the tree");

for(Position<String> p : LBT10.inOrder())

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

System.out.println("\n\nThe postOrder traversal of the tree");

for(Position<String> p : LBT10.postOrder())

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

System.out.println("\n\nThe breathFirst traversal of the tree");

for(Position<String> p : LBT10.breadthfirst())

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

System.out.println("\n\nThe preOrderIndent traversal of the tree");

printPreOrderIndent(LBT10, LBT10.root, 0);

System.out.println("\n\nA The Parenthesized representation of the tree");

parenthesize(LBT10, LBT10.root);

System.out.println("\n\nEnd Lab108");

}

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

int d = path.size();

System.out.print(spaces(2\*d));

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

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

path.add(1);

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

printPreOrderLabeled(T, c, path);

path.set(d, 1 + path.get(d));

}

path.remove(d);

}

private static String spaces(int t){

String temp = "";

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

temp +=(" ");

return temp;

}

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

int subtotal = p.getElement();

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

subtotal += diskSpace(T, c);

return subtotal;

}

public static <E> void parenthesize(TreeInterface<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 ? " (" : ", "));

firstTime = false;

parenthesize(T, c);

}

System.out.print(")");

}

}

public static <E> void printPreOrderIndent(TreeInterface<E> T, Position<E> p, int d){

System.out.println(spaces(2 \* d) + p.getElement());

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

printPreOrderIndent(T, c, d+1);

}

}

**AbstractBinaryTree**

import java.util.Iterator;

import java.util.ArrayList;

import java.util.List;

/\*\*

\*

\* @author Marie Larson

\* @version 3/21/18

\* @param <E>

\*/

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

@Override

//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 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)

}

@Override

//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;

}

@Override

//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 (Iterable<Position<E>>) snapshot;

}

//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 (Position<E> c : children(p)){

preorderSubtree(c, snapshot);

}

}

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

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

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

if(!isEmpty()){

preorderSubtree(root(), snapshot);

}

return (Iterable<Position<E>>) 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);

}

//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);

return(Iterable<Position<E>>) snapshot;

}

//Returns an iterable colection of positions for 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<Position<E>>();

fringe.enqueue(root());

while(!fringe.isEmpty()){

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

snapshot.add(p);

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

fringe.enqueue(c);

}

}

return snapshot;

}

//Adds positions of the subtree rooted at Postion 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);

return snapshot;

}

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

@Override

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

return inOrder();

}

}

**AbstractTree**

import java.util.ArrayList;

import java.util.List;

import java.util.Iterator;

/\*\*

\*

\* @author Marie Larson

\* @version 3/21/18

\*/

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

@Override

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

@Override

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

@Override

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

//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 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;

}

}

**BinaryTreeInterface**

/\*\*

\*

\* @author Marie Larson

\* @version 3/21/18

\*/

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

public interface BinaryTreeInterface<E> extends TreeInterface<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**

import java.util.Iterator;

/\*\*

\*

\* @author Marie Larson

\* @version 3/21/18

\* @param <E>

\*/

//Concrete implemenatation 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 stroed 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

@Override

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 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 i the tree

//constructor

public LinkedBinaryTree(){} //contructs 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 positon type");

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

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

throw new IllegalArgumentException("p is not longer in the tree");

return node;

}

//accessor methods (not already implemented in AbstractBinaryTree)

//Returns the number of nodes in the tree.

@Override

public int size(){

return size;

}

//returns the root position of the tree (or null if tree is empty).

@Override

public boolean isEmpty(){

return size==0;

}

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

private class ElementIterator implements Iterator<E>{

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

@Override

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

@Override

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

@Override

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

}

//returns an iterator of the elements stored in tree.

@Override

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

public Iterable<Position<E>> postions(){return preOrder();}

@Override

//returns the root position of the tree(null if the is empty).

public Position<E> root(){

return root;

}

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

@Override

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

Node<E> node = validate(p);

return node.getParent();

}

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

@Override

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

Node<E> node = validate(p);

return node.getLeft();

}

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

@Override

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

Node<E> node = validate(p);

return node.getRight();

}

//update methods supported by this class

//Places element e at the root of an empty tree and returns its new Position.

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

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

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

//parent.setLeft(child);

size++;

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 tree 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------------------

**TreeInterface**

import java.util.Iterator;

/\*\*

\*

\* @author Marie Larson

\* @version 3/21/18

\*/

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

public interface TreeInterface<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();

@Override

Iterator<E> iterator();

Iterable<Position<E>> positions();

}

**Output:**

run:

Begin Lab108

Creating the following equation in a linkedBinaryTree: (((5+2)\*(2-1)/(2+9)+((7-2)-1))\*8)

A The preoder traversal of the tree

\*+/\*+52-21+298

The inOrder traversal of the tree

5+2\*2-1/2+9+\*8

The postOrder traversal of the tree

52+21-\*29+/+8\*

The breathFirst traversal of the tree

\*+8/\*++-295221

The preOrderIndent traversal of the tree

\*

+

/

\*

+

5

2

-

2

1

+

2

9

8

A The Parenthesized representation of the tree

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

End Lab108

BUILD SUCCESSFUL (total time: 0 seconds)