**Tree**

import java.util.Iterator;

/\*\*

\*

\* @author Goodrich, Tamassia, Goldwasser

\*/

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

/\*\*

\*

\* @return the position of the root of the tree or null if empty.

\*/

Position<E> root();

/\*\*

\*

\* @param p position

\* @return the position of the parent of position p or null if p is the root

\*/

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

/\*\*

\*

\* @param p

\* @return an itreable collection containing the children of position p (if any)

\* @throws IllegalArgumentException

\*/

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

/\*\*

\*

\* @param p

\* @return the number of children of position p

\*

\*/

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

/\*\*

\*

\* @return true if positio p has at least one child

\*/

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

/\*\*

\*

\* @param p

\* @return treue if posion p does not have any children

\*/

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

/\*\*

\*

\* @param p

\* @return true if position p is the root of the tree

\*/

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

/\*\*

\*

\* @return the number of positions(and hence elements)

\* that are contained in the tree.

\*/

int size();

/\*\*

\*

\* @return true if the tree does not contain any positions.

\*/

boolean isEmpty();

/\*\*

\*

\* @return an iterator for all elements in the tree

\* (so that the tree itself is Iterable).

\*/

Iterator<E> iterator();

/\*\*

\*

\* @return an iterable collection of all position of the tree.

\*/

Iterable<Position<E>> positions();

}

**AbstractBinaryTree Class**

import java.util.ArrayList;

import java.util.List;

/\*\*

\* An abstract base class providing some functionality

\* of the Binary Tree Interface.

\* @author Goodrich, Tamassia, Goldwasser

\*/

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

/\*\* return 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;

if (p == left(parent))

return right(parent);

else

return left(parent);

}

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);

if(left(p) != null)

snapshot.add(left(p));

if(right(p)!= null)

snapshot.add(right(p));

return snapshot;

}

// code fragement 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 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. \*/

public Iterable<Position<E>> positions()

{

return inorder();

}

//code fragment 8.23

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); //child depth is d+1

}

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

{

int d = path.size();

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

for(int j =0; j<d; j++)

System.out.print(path.get(j) +(j==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)); //increment last entry of path

}

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

}

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 ? " (" : ", "));

firstTime = false;

parenthesize(T,c);

}

System.out.print(")");

}

}

// utility method

protected static String spaces(int d)

{

String spaceWidth = " ";

for(int i = 0; i < d; i++ )

spaceWidth += " ";

return spaceWidth;

}

}

**BinaryTree Interface**

/\*\*

\* An interface for a binary tree, in which each node has

\* at most two children.

\* @author Goodrich, Tamassia, Goldwasser

\*/

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

/\*\*

\*

\* @param p

\* @return the position of p's left child (or null if no child exist)

\* @throws IllegalArgumentException

\*/

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

/\*\*

\*

\* @param p

\* @return the position of p's right child

\* @throws IllegalArgumentException

\*/

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

/\*\*

\*

\* @param p

\* @return the position of the sibling of p or null

\* @throws IllegalArgumentException

\*/

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

}

**LinkedBinaryTree Class**

/\*\*

\*

\* @author Rich

\*/

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 refence to the right child (if any)

/\*\* Construct 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);

}

//LinkedBinaryTreee instance variables

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

private int size = 0; // nummber of nodes in the three

//constructor

public LinkedBinaryTree(){} // construct an empty birnary 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 alread 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 exist) \*/

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

{

Node<E> node = validate(p);

return node.getLeft();

}

/\*\* Retruns the Positon 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 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 alread has a left child");

Node<E> child = createNode(e, parent,null, null);

parent.setLeft(child);

size++;

return child;

}

/\*\* Create 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;

}

/\*\* Replace 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 tress t1 and t2 as left and right subtrees of exteranal 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);

node.setLeft(t1.root);

t1.root= null;

t1.size = 0;

}

if(!t2.isEmpty())

{

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 granparent becomes its parent

if(node == root)

root = child;

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);

node.setLeft(null);

node.setRight(null);

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

return temp;

}

}

**Queue Interface**

/\*\*

\*

\* @author Goodrich, Tamassia, Goldwasser

\*/

public interface Queue<E> {

/\*\*

\*

\* @return the number of element in the queue.

\*/

int size();

/\*\*

\*

\* @return true if the queue is empty

\*/

boolean isEmpty();

/\*\*

\*

\* @param e element to be inserted at the rear of the queue

\*/

void enqueue(E e);

/\*\*

\*

\* @return but does not remove, the first element of the queue or null if empty

\*/

E first();

/\*\*

\* Remove the first element of the queue(Null if empty)

\* @return the element removed

\*/

E dequeue();

}

**StinglyLinkedList(From old lab) Class**

/\*\*

\*

\* @author Rich

\* @version 03/16/2017

\* This class consist of method for creations of nodes, and methods on how to manipulate nodes

\* data structure.

\*/

public class SinglyLinkedList<E> {

private static class Node<E>{

private E element;

private Node<E> next;

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

{

element = e;

next = n;

}

/\*\*

\*

\* @return the element in the node.

\*/

public E getElement()

{

return element;

}

/\*\*

\*

\* @return the pointer to the next node of the list.

\*/

public Node<E> getNext()

{

return next;

}

// setters

/\*\*

\*

\* @param newNext set the pointer to point to the next item

\*/

public void setNext(Node<E> newNext)

{

next = newNext;

}

}

// continuing of the StinglyLinkedList

private Node<E> head = null; // point to 1st node of the list

private Node<E> tail = null; // pointer to the last node of the list

private int size = 0;

public SinglyLinkedList(){} // construcst an initially empty list.

/\*\*

\*

\* @return how many nodes in the list

\*/

public int size()

{

return size;

}

/\*\*

\*

\* @return true if list is empty.

\*/

public boolean isEmpty()

{

return size==0;

}

/\*\*

\*

\* @return the first element in the first node

\*/

public E first()

{

if (isEmpty())

return null;

return head.getElement();

}

/\*\*

\*

\* @return the last element in the list

\*/

public E last()

{

if(isEmpty()) return null;

return tail.getElement();

}

/\*\*

\*

\* @param e generic element to be place as first element

\*/

public void addFirst(E e)

{

head = new Node(e,head);

if (size == 0)

tail = head;

size++;

}

/\*\*

\*

\* @param e generic type element to set as last

\*/

public void addLast(E e)

{

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

if(isEmpty())

{

head = newest;

}

else

tail.setNext(newest);

tail = newest;

size++;

}

/\*\*

\*

\* @return the element in the 1st node removed

\*/

public E removeFirst()

{

if (isEmpty())

return null;

E answer = head.getElement();

head = head.getNext();

size--;

if(size == 0)

tail = null;

return answer;

}

// my removeLast

/\*\*

\*

\* @return the removed item

\*/

public E removeLast()

{

if(isEmpty()) return null;

E answer = tail.getElement();

Node<E> current = head, previous = head;

while(current.getNext() != null)

{

previous = current; // the one before to last node/tail.

current = current.getNext();

}

// after exiting while loop current holds the memRef of tail, it is pointing to tail. current =tail

previous.setNext(null); // break the chain btw the one- before last and last node.

tail = previous;

return answer;

}

/\*\*

\*

\* @return String format of the object

\*/

@Override

public String toString()

{

String listElements = "";

Node<E> current = head;

while(current != null)

{

listElements += current.getElement() +"-->";

current = current.getNext(); // update current to point to next node in the listt.

}

return listElements;

}

/\*\*

\*

\* @param o object ref

\* @return true if two linked list are equal

\*/

public boolean equals(Object o)

{

if(!(o instanceof SinglyLinkedList))

return false;

SinglyLinkedList l = (SinglyLinkedList) o;

if (size != l.size())

return false;

Node<E> sourceCurrentNodePtr = head; // current node pointer/Refvar for the "blueprint" list.

Node<E> targetCurrentNodePtr = l.head; // identifiers that points to current head of the list we're testing for equality.

while(sourceCurrentNodePtr != null)

{

if(!sourceCurrentNodePtr.getElement().equals(targetCurrentNodePtr.getElement()))

return false;

sourceCurrentNodePtr = sourceCurrentNodePtr.getNext(); // updtate memory pointer.; advancing current to next Node.

targetCurrentNodePtr = targetCurrentNodePtr.getNext();

}

return true;

}

}

**LinkedQueue**

/\*\*

\*

\* @author Goodrich, Tamassia, Goldwasser

\*/

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

// create an emtpy queue/list

private SinglyLinkedList<E> list = new SinglyLinkedList<>();

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();}

}

**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;

}

**Client Class**

import java.util.Iterator;

/\*\*

\*

\* @author Richelin Metellus

\* @version 03/10/2017

\* Printing of different ways to visit a tree.

\*/

public class TreeClient {

public static void main(String[] args) {

LinkedBinaryTree<String>[] lbt = new LinkedBinaryTree[19];

for(int i = 0; i<19; i++)

lbt[i] = new LinkedBinaryTree();

Position[] positions = new Position[19];

positions[0] = lbt[0].addRoot("\*");

positions[1] = lbt[1].addRoot("/");

positions[2] = lbt[2].addRoot("9");

positions[3] = lbt[3].addRoot("\*");

positions[4] = lbt[4].addRoot("+");

positions[5] = lbt[5].addRoot("+");

positions[6] = lbt[6].addRoot("-");

positions[7] = lbt[7].addRoot("+");

positions[8] = lbt[8].addRoot("-");

positions[9] = lbt[9].addRoot("7");

positions[10] = lbt[10].addRoot("5");

positions[11] = lbt[11].addRoot("9");

positions[12] = lbt[12].addRoot("3");

positions[13] = lbt[13].addRoot("15");

positions[14] = lbt[14].addRoot("24");

positions[15] = lbt[15].addRoot("-");

positions[16] = lbt[16].addRoot("5");

positions[17] = lbt[17].addRoot("6");

positions[18] = lbt[18].addRoot("1");

// attaching the subtree in bottom up fashion

lbt[15].attach(positions[15], lbt[17], lbt[18]);

lbt[5].attach(positions[5], lbt[9], lbt[10]);

lbt[6].attach(positions[6],lbt[11],lbt[12]);

lbt[7].attach(positions[7], lbt[13],lbt[14]);

lbt[8].attach(positions[8],lbt[15], lbt[16]);

lbt[3].attach(positions[3],lbt[5],lbt[6]);

lbt[4].attach(positions[4], lbt[7],lbt[8]);

lbt[1].attach(positions[1],lbt[3],lbt[4]);

lbt[0].attach(positions[0],lbt[1],lbt[2]);

System.out.println(" The experession: (((7+5)\*(9-3))/((15+24)+((6-1)-5))\*9)");

// Inorder

Iterator<Position<String>> inOrderIterator = lbt[0].inorder().iterator();

while(inOrderIterator.hasNext())

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

System.out.println(" Inorder Traversal of the Tree\n");

Iterator<Position<String>> preOrderIterator = lbt[0].preorder().iterator();

while(preOrderIterator.hasNext())

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

System.out.println(" preOrder traversal of the Tree \n");

//Postorder traversal

Iterator<Position<String>> postOrderIterator = lbt[0].postorder().iterator();

while(postOrderIterator.hasNext())

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

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

Iterator<Position<String>> breathFirstIterator = lbt[0].breathFirst().iterator();

while(breathFirstIterator.hasNext())

System.out.print(breathFirstIterator.next().getElement()+ " ");

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

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

AbstractBinaryTree.printPreorderIndent(lbt[0], lbt[0].root, 0);

System.out.println("Parenthesize representation of the tree");

AbstractBinaryTree.parenthesize(lbt[0], positions[0]);

System.out.println("");

}

}

**Program Output**

**run:**

**The experession: (((7+5)\*(9-3))/((15+24)+((6-1)-5))\*9)**

**7+5\*9-3/15+24+6-1-5\*9 Inorder Traversal of the Tree**

**\* / \* + 7 5 - 9 3 + + 15 24 - - 6 1 5 9 preOrder traversal of the Tree**

**7 5 + 9 3 - \* 15 24 + 6 1 - 5 - + / 9 \* postOrder traversal of the Tree**

**\* / 9 \* + + - + - 7 5 9 3 15 24 - 5 6 1 breathFirst traversal of the Tree**

**preOrderIndent traversal of the tree**

**\***

**/**

**\***

**+**

**7**

**5**

**-**

**9**

**3**

**+**

**+**

**15**

**24**

**-**

**-**

**6**

**1**

**5**

**9**

**Parenthesize representation of the tree**

**\* (/ (\* (+ (7, 5), - (9, 3)), + (+ (15, 24), - (- (6, 1), 5))), 9)**

**BUILD SUCCESSFUL (total time: 1 second)**