Task1:



Arithmetic:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package Arithmetic;

import GFile.\*;

import Stack.LinkedStack;

import Tree.Expression;

import Tree.LinkedBinaryTree;

import java.io.File;

import java.util.ArrayList;

import java.util.Scanner;

/\*\*

\* Client class that tests the functionality of the Tree package and EvaluateBinaryTree class

\* @author Jacob Huesman

\*/

public class Arithmetic {

public static void main(String args[]) throws Exception{

/\* Instance Variables \*/

Scanner scan = new Scanner(System.in);

Scanner file;

String[] files;

ArrayList<LinkedBinaryTree> trees = new ArrayList<>();

LinkedStack<String> temp = new LinkedStack<>();

/\* Get files from user \*/

try {

System.out.println("Please enter your files...");

files = GFile.getFiles();

System.out.println("List of files selected:");

for (String file1 : files) {

System.out.println(file1);

}

/\* Loop through files \*/

for (String filePath : files) {

file = new Scanner(new File(filePath));

String line;

System.out.println("\nCurrent file: " + filePath);

/\* Process each line of the file as an expression \*/

int exp = 1;

while(file.hasNextLine()){

try {

line = file.nextLine();

System.out.println("\n" + exp + ") Current Expression: " + line);

LinkedBinaryTree tree = Expression.makeTree(line);

printOrders(tree);

} catch (Exception e) {

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

}

exp++;

}

}

} catch (Exception e) {

System.out.println(e);

}

}

/\*\*

\* Print the tree in preorder, inorder and postorder notation. This method also prints the value of the evaluated expression.

\* @param tree Tree to be evaluated

\*/

private static void printOrders(LinkedBinaryTree tree){

System.out.println("Preorder:");

LinkedBinaryTree.printPreorderIndent(tree, tree.root(), 0);

System.out.println("Inorder:");

tree.printInorderIndentSubtree(tree.root(), 0);

System.out.println("Postorder:");

LinkedBinaryTree.printPostorderIndent(tree, tree.root(), 0);

System.out.println("Value: " + EvaluateBinaryTree.evaluate(tree, tree.root()));

}

}

EvaluateBinaryTree:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package Arithmetic;

import Tree.Position;

import Tree.Tree;

import java.util.Iterator;

/\*\*

\* Evaluates a binary expression tree

\* @author Jacob Huesman

\*/

public class EvaluateBinaryTree {

public static int evaluate(Tree<String> T, Position<String> p){

if(isExpression(p.getElement())){

Iterator children = T.children(p).iterator();

Position<String> left = (Position) children.next();

Position<String> right = (Position) children.next();

switch(p.getElement().charAt(0)){

case '\*' :

case 'x' :

return evaluate(T, left) \* evaluate(T, right);

case '/' :

return evaluate(T, left) / evaluate(T, right);

case '+' :

return evaluate(T, left) + evaluate(T, right);

case '-' :

return evaluate(T, left) - evaluate(T, right);

}

}

return Integer.parseInt(p.getElement());

}

/\*\*

\* Checks a String to see if it represents a simple Arithmetic expression

\* @param exp String to be evaluated

\* @return true - If exp is an expression; false - otherwise

\*/

private static boolean isExpression(String exp){

String[] exps = {"\*","x","+","-","/"};

for(String tExp : exps){

if(tExp.equals(exp)){

return true;

}

}

return false;

}

}

FilePath:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package GFile;

import java.io.File;

import java.util.ArrayList;

/\*\*

\* Simple class designed to hold the paths to files for storage across multiple threads.

\* @author Jacob Huesman

\*/

public class FilePath {

private ArrayList<String> filePath;

private boolean hasPath;

/\*\*

\* Default constructor.

\*/

public FilePath(){

filePath = new ArrayList(1);

hasPath = false;

}

/\*\*

\* Adds a file path to the FilePath object. Before adding it checks to ensure the file path is valid.

\* @param filePath String representation of the file path

\* @return true if the file path passed as a parameter was valid and added; false - otherwise

\*/

public boolean addPath(String filePath){

if(new File(filePath).isFile()){

this.filePath.add(filePath);

hasPath = true;

return true;

} else {

return false;

}

}

/\*\*

\* Returns a string array of file paths.

\* @return String[] representing the file paths.

\*/

public String[] getPaths(){

String[] str = new String[filePath.size()];

str = filePath.toArray(str);

return str;

}

/\*\*

\* Indicates if the object contains at least one file path.

\* @return true - if there is at least one file path contained in this object; false - otherwise

\*/

public boolean hasPath() {

return hasPath;

}

}

GFile:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package GFile;

/\*\*

\* Class containing simple methods to graphically request files from a user.

\* @author Jacob Huesman

\*/

public class GFile {

public static String[] getFiles() throws Exception{

FilePath path = new FilePath();

//Makes an empty thread that causes the the main client to wait until the thread finishes executing. The thread finishes executing when the user either selects a file or closes the window.

Thread thread = new Thread(new Runnable(){public void run(){try {while(true){Thread.sleep(1000);}} catch (InterruptedException e){}}});

thread.start();

//Runs an instance of GetFile. GetFile is a GUI that prompts the user for a file to import data from.

java.awt.EventQueue.invokeLater(new Runnable() {

public void run() {

new UI(path, thread).setVisible(true);

}

});

//The client will wait for the user to select a file before continuing.

thread.join();

//Checks if the user provided a file. Triggered if user exits the application without importing a file.

if(!path.hasPath()){

throw new Exception("User failed to provide a valid file.");

}

//Return the file path

return path.getPaths();

}

}

UI:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package GFile;

import javax.swing.JFileChooser;

import javax.swing.UIManager;

/\*\*

\* Simple GUI for selecting several files.

\* @author Jacob Huesman

\*/

public class UI extends javax.swing.JFrame {

/\*\*

\* Declare instance variables.

\*/

private final JFileChooser fc;

private final FilePath file;

private final Thread thread;

/\*\*

\* Creates new JFrame GetFile.

\* @param filePath object used to store file paths across multiple threads

\* @param thread this thread prevents the GFile method from continuing execution until the file selection process is complete

\*/

public UI(FilePath filePath, Thread thread) {

/\* Set the System look and feel \*/

//<editor-fold defaultstate="collapsed" desc=" Look and feel setting code (optional) ">

try {

UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());

} catch (ClassNotFoundException ex) {

java.util.logging.Logger.getLogger(UI.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);

} catch (InstantiationException ex) {

java.util.logging.Logger.getLogger(UI.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);

} catch (IllegalAccessException ex) {

java.util.logging.Logger.getLogger(UI.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);

} catch (javax.swing.UnsupportedLookAndFeelException ex) {

java.util.logging.Logger.getLogger(UI.class.getName()).log(java.util.logging.Level.SEVERE, null, ex);

}

//</editor-fold>

initComponents();

/\*\*

\* Instantiate new JFileChooser. Add references of filePath and thread passed from the client.

\*/

fc = new JFileChooser();

this.file = filePath;

this.thread = thread;

}

/\*\*

\* This method is called from within the constructor to initialize the form.

\* WARNING: Do NOT modify this code. The content of this method is always

\* regenerated by the Form Editor.

\*/

@SuppressWarnings("unchecked")

// <editor-fold defaultstate="collapsed" desc="Generated Code">

private void initComponents() {

jScrollPane1 = new javax.swing.JScrollPane();

jTextArea1 = new javax.swing.JTextArea();

filePath = new javax.swing.JTextField();

findFile = new javax.swing.JButton();

returnFile = new javax.swing.JButton();

label = new javax.swing.JLabel();

jScrollPane2 = new javax.swing.JScrollPane();

filePaths = new javax.swing.JTextArea();

add = new javax.swing.JButton();

jTextArea1.setEditable(false);

jTextArea1.setColumns(20);

jTextArea1.setRows(5);

jTextArea1.setAutoscrolls(false);

jScrollPane1.setViewportView(jTextArea1);

setDefaultCloseOperation(javax.swing.WindowConstants.DO\_NOTHING\_ON\_CLOSE);

addWindowListener(new java.awt.event.WindowAdapter() {

public void windowClosing(java.awt.event.WindowEvent evt) {

formWindowClosing(evt);

}

});

filePath.setToolTipText("");

findFile.setText("File");

findFile.addActionListener(new java.awt.event.ActionListener() {

public void actionPerformed(java.awt.event.ActionEvent evt) {

findFileActionPerformed(evt);

}

});

returnFile.setText("Import");

returnFile.addActionListener(new java.awt.event.ActionListener() {

public void actionPerformed(java.awt.event.ActionEvent evt) {

returnFileActionPerformed(evt);

}

});

label.setText("Enter a file path and click Add, or click File and choose the file you want to add. When done click import.");

filePaths.setEditable(false);

filePaths.setColumns(20);

filePaths.setRows(5);

jScrollPane2.setViewportView(filePaths);

add.setText("Add");

add.addActionListener(new java.awt.event.ActionListener() {

public void actionPerformed(java.awt.event.ActionEvent evt) {

addActionPerformed(evt);

}

});

javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());

getContentPane().setLayout(layout);

layout.setHorizontalGroup(

layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addGroup(layout.createSequentialGroup()

.addContainerGap()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addGroup(layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING, false)

.addComponent(jScrollPane2)

.addComponent(label, javax.swing.GroupLayout.DEFAULT\_SIZE, 508, Short.MAX\_VALUE))

.addGap(0, 0, Short.MAX\_VALUE))

.addGroup(layout.createSequentialGroup()

.addComponent(filePath)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addComponent(add, javax.swing.GroupLayout.PREFERRED\_SIZE, 65, javax.swing.GroupLayout.PREFERRED\_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addComponent(findFile, javax.swing.GroupLayout.PREFERRED\_SIZE, 65, javax.swing.GroupLayout.PREFERRED\_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addComponent(returnFile, javax.swing.GroupLayout.PREFERRED\_SIZE, 65, javax.swing.GroupLayout.PREFERRED\_SIZE)))

.addContainerGap())

);

layout.setVerticalGroup(

layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)

.addGroup(javax.swing.GroupLayout.Alignment.TRAILING, layout.createSequentialGroup()

.addContainerGap()

.addComponent(label)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addComponent(jScrollPane2, javax.swing.GroupLayout.PREFERRED\_SIZE, javax.swing.GroupLayout.DEFAULT\_SIZE, javax.swing.GroupLayout.PREFERRED\_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)

.addComponent(filePath, javax.swing.GroupLayout.PREFERRED\_SIZE, javax.swing.GroupLayout.DEFAULT\_SIZE, javax.swing.GroupLayout.PREFERRED\_SIZE)

.addComponent(findFile)

.addComponent(returnFile)

.addComponent(add))

.addContainerGap(javax.swing.GroupLayout.DEFAULT\_SIZE, Short.MAX\_VALUE))

);

pack();

}// </editor-fold>

/\*\*

\* Opens JFileChooser dialog and then displays and stores the path.

\* @param evt

\*/

private void findFileActionPerformed(java.awt.event.ActionEvent evt) {

if(fc.showOpenDialog(this) == JFileChooser.APPROVE\_OPTION){

file.addPath(fc.getSelectedFile().getAbsolutePath());

filePaths.setText(filePaths.getText()+fc.getSelectedFile().getAbsolutePath()+"\n");

}

}

/\*\*

\* Closes the dialog.

\* @param evt

\*/

private void returnFileActionPerformed(java.awt.event.ActionEvent evt) {

thread.interrupt();

this.dispose();

}

/\*\*

\* On window close notify thread to stop executing. Then dispose of this Frame's resources.

\* @param evt

\*/

private void formWindowClosing(java.awt.event.WindowEvent evt) {

thread.interrupt();

this.dispose();

}

/\*\*

\* Add the file path to the list of file paths.

\* @param evt

\*/

private void addActionPerformed(java.awt.event.ActionEvent evt) {

if(file.addPath(filePath.getText())){

filePaths.setText(filePaths.getText()+filePath.getText()+"\n");

} else {

filePath.setText("Please enter a valid file path");

}

}

// Variables declaration - do not modify

private javax.swing.JButton add;

private javax.swing.JTextField filePath;

private javax.swing.JTextArea filePaths;

private javax.swing.JButton findFile;

private javax.swing.JScrollPane jScrollPane1;

private javax.swing.JScrollPane jScrollPane2;

private javax.swing.JTextArea jTextArea1;

private javax.swing.JLabel label;

private javax.swing.JButton returnFile;

// End of variables declaration

}

LinkedStack:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package Stack;

/\*\*

\* SLL implementation of the stack data structure.

\* @author Jacob Huesman

\* @param <T> Type of objects to be contained in the stack

\*/

public class LinkedStack <T> {

/\*\*

\* The objects of this class form the nodes of the singly linked list.

\* @param <T> Type of the object the node will hold

\*/

private static class Node<T> {

private final T element;

private Node<T> next;

public Node(T e, Node<T> n){

element = e;

next = n;

}

public T getElement(){ return element; }

public Node<T> getNext() { return next; }

public void setNext( Node<T> n ){ next = n; }

}

private Node<T> stackHead;

int count;

/\*\*

\* Default constructor.

\*/

public LinkedStack(){

stackHead = null;

count = 0;

}

/\*\*

\* Returns a count of the items in the stack.

\* @return A count of items in stack

\*/

public int getCurrentSize() {

return count;

}

/\*\*

\* Checks if the stack is empty.

\* @return true - if empty; false - otherwise

\*/

public boolean isEmpty() {

return count <= 0;

}

/\*\*

\* Adds an element to the beginning of the list.

\* @param item The item the node will point to

\* @return true - when the operation is successful

\*/

public boolean push(T item) {

stackHead = new Node(item, stackHead);

count++;

return true;

}

/\*\*

\* Removes and returns the first node in the list.

\* @return The element, or null if the list is empty

\*/

public T pop() {

if(stackHead == null){

return null;

}

T element = stackHead.getElement();

stackHead = stackHead.getNext();

count--;

return element;

}

/\*\*

\* Clears the stack.

\*/

public void clear() {

stackHead = null;

count = 0;

}

/\*\*

\* Returns the first item in the stack.

\* @return First item in the stack

\*/

public T first(){

if (stackHead!= null){

return stackHead.getElement();

}

return null;

}

}

AbstractBinaryTree:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package Tree;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* AbstractBinaryTree class outlined in Data Structures & Algorithms by Michael T Goodrich, Roberto Tamassia, and Michael H Goldwasser.

\* @author Jacob Huesman

\* @param <E>

\*/

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

/\*\*

\* Returns the Position of p's sibling (or null if no sibling exists)

\* @param p A valid Position within the tree

\* @return the Position of the sibling (or null if no sibling exists)

\* @throws IllegalArgumentException if p is not a valid Position for this tree

\*/

@Override

public Position<E> sibling(Position<E> p) {

Position<E> parent = parent(p);

if(parent == null){

return null;

} else if(parent == left(parent)){

return right(parent);

} else {

return left(parent);

}

}

/\*\*

\* Returns the number of children of a given node

\* @param p the node

\* @return number of children of the node

\* @throws IllegalArgumentException

\*/

@Override

public int numChildren(Position<E> p){

int count = 0;

if(left(p) != null){

count++;

}

if(right(p) != null){

count++;

}

return count;

}

/\*\*

\* Returns an iterator containing the positions of the children of the node

\* @param p the node

\* @return iterator containing positions of the node's children

\* @throws IllegalArgumentException

\*/

@Override

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;

}

/\*\*

\* Adds positions of the subtree rooted at Position p to the given snapshot using an inorder traversal

\* @param p Position at the root of a subtree

\* @param snapshot list where the results are added

\*/

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 in order.

\* @return iterable collection of the tree's positions

\*/

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

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

if (!isEmpty()){

inorderSubtree(root(), snapshot); // fill the snapshot recursively

}

return snapshot;

}

/\*\*

\* Returns an iterable collection of the positions of the tree using inorder traversal

\* @return iterable collection of the tree's positions

\*/

@Override

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

return inorder();

}

/\*\*

\* Adds positions of the subtree rooted at Position p to the given snapshot.

\* @param p Position at the root of the subtree

\* @param snapshot List of the positions

\*/

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 of the tree, reported in preorder

\* @return iterable collection of positions of the tree

\*/

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

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

if(!isEmpty()){

preorderSubtree(root(), snapshot);

}

return snapshot;

}

/\*\*

\* Adds positions of the subtree rooted at Position p to the given snapshot

\* @param p Position at the root of the subtree

\* @param snapshot List of the positions

\*/

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

\* @return iterable collection of positions of the tree

\*/

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

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

if(!isEmpty()){

postorderSubtree(root(), snapshot);

}

return snapshot;

}

/\*\*

\* Recursively prints the positions of a tree in preOrder.

\* @param <E> Type of tree

\* @param T The tree to be printed in preOrder

\* @param p The starting position in the tree

\* @param d The current depth

\*/

public static <E> void printPreorderIndent(Tree<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);

}

}

/\*\*

\* Recursively prints the positions of a tree in postOrder.

\* @param <E> Type of tree

\* @param T The tree to be printed in preOrder

\* @param p The starting position in the tree

\* @param d The current depth

\*/

public static <E> void printPostorderIndent(Tree<E> T, Position<E> p, int d){

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

printPostorderIndent(T, c, d+1);

}

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

}

/\*\*

\* Recursively prints the positions of a tree in inOrder.

\* @param p The starting position in the tree

\* @param d The current depth

\*/

public void printInorderIndentSubtree(Position<E> p, int d){

if(left(p) != null){

printInorderIndentSubtree(left(p), (d+1));

}

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

if(right(p) != null){

printInorderIndentSubtree(right(p), (d+1));

}

}

/\*\*

\* Simple function to return a string of a desired number of spaces

\* @param n Desired number of spaces

\* @return String of the desired number of spaces

\*/

private static String spaces(int n){

char[] spaces = new char[n];

StringBuilder space = new StringBuilder();

for(int i=0; i<spaces.length; i++){

spaces[i] = ' ';

}

space.append(spaces);

return space.toString();

}

}

AbstractTree:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package Tree;

import java.util.Iterator;

/\*\*

\* AbstractTree class outlined in Data Structures & Algorithms by Michael T Goodrich, Roberto Tamassia, and Michael H Goldwasser.

\* @author Jacob Huesman

\* @param <E>

\*/

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

/\*\*

\* Returns if the node is Internal

\* @param p the position of the node

\* @return true - if the node is internal; false - otherwise

\* @throws IllegalArgumentException

\*/

@Override

public boolean isInternal(Position<E> p){

return numChildren(p) > 0;

}

/\*\*

\* Returns if the node is External

\* @param p the position of the node

\* @return true - if the node is external; false - otherwise

\* @throws IllegalArgumentException

\*/

@Override

public boolean isExternal(Position<E> p){

return numChildren(p) == 0;

}

/\*\*

\* Returns if the node is the root of the tree

\* @param p the position of the node

\* @return true - if the node is the root; false - otherwise

\* @throws IllegalArgumentException

\*/

@Override

public boolean isRoot(Position<E> p){

return p == root();

}

/\*\*

\* Returns if the tree is empty

\* @return true - if the tree is empty; false - otherwise

\*/

@Override

public boolean isEmpty(){

return size() == 0;

}

/\*\*

\* Returns the number of levels separating Position p from the root

\* @param p A valid Position within the tree

\* @return 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

\* @param p A valid Position within the tree

\* @return 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;

}

/\* 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 for the tree

\* @return Iterator

\*/

@Override

public Iterator<E> iterator(){

return new ElementIterator();

}

}

BinaryTree:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package Tree;

/\*\*

\* Binary tree class outlined in Data Structures & Algorithms by Michael T Goodrich, Roberto Tamassia, and Michael H Goldwasser.

\* @author Jacob Huesman

\*/

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

/\*\*

\* Returns the positions left child (or null if no child exists)

\* @param p A position within the tree

\* @return the Positions of the left child (or null if no child exists)

\* @throws IllegalArgumentException if p is not a valid Position for this tree

\*/

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

/\*\*

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

\* @param p A position within the tree

\* @return the Position of the right child (or null if no child exists)

\* @throws IllegalArgumentException if p is not a valid Position for this tree

\*/

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

/\*\*

\* Returns the Position of p's sibling (or null if no sibling exists)

\* @param p A valid Position within the tree

\* @return the Position of the sibling (or null if no sibling exists)

\* @throws IllegalArgumentException if p is not a valid Position for this tree

\*/

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

}

Expression:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package Tree;

import Tree.LinkedBinaryTree.Node;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* Takes a List of String tokens and creates a binary expression tree from them

\* @author Jacob Huesman

\*/

public class Expression {

/\*\*

\* Using a combination of looping and recursion this method takes a List of String tokens and creates a binary expression tree from them

\* @param tokens A List of Strings that represent an arithmetic expression

\* @return A LinkedBinaryTree representation of the expression

\* @throws Exception If the expression is invalid or unbalanced

\*/

public static LinkedBinaryTree<String> makeTree(List<String> tokens) throws Exception{

ArrayList<Node<String>> nodes = new ArrayList<>();

/\* Recursively evaluate functions in parentheses \*/

for(int i=0; i<tokens.size(); i++){

if(tokens.get(i).contains("(")){

int parentheses = 0;

for(int j=i+1; j<tokens.size(); j++){

if((tokens.get(j).equals(")")) && (parentheses == 0)){

nodes.add(makeTree(tokens.subList(i+1, j)).root);

i=j;

break;

} else if(tokens.get(j).equals("(")){

parentheses++;

} else if(tokens.get(j).equals(")")){

parentheses--;

}

}

} else {

nodes.add(new Node<>(tokens.get(i), null, null, null));

}

}

/\* Check for \*,x,/ operators \*/

for(int i=0; i<nodes.size(); i++){

if(nodes.get(i).getLeft() == null && nodes.get(i).getRight() == null && (nodes.get(i).getElement().equals("\*") || nodes.get(i).getElement().equals("x") || nodes.get(i).getElement().equals("/"))){

Node<String> root = nodes.get(i);

root.setLeft(nodes.get(i-1));

root.setRight(nodes.get(i+1));

nodes.remove(i-1);

nodes.remove(i);

i--;

}

}

/\* Check for +, - operators \*/

for(int i=0; i<nodes.size(); i++){

if(nodes.get(i).getLeft() == null && nodes.get(i).getRight() == null && (nodes.get(i).getElement().equals("+") || nodes.get(i).getElement().equals("-"))){

Node<String> root = nodes.get(i);

root.setLeft(nodes.get(i-1));

root.setRight(nodes.get(i+1));

nodes.remove(i-1);

nodes.remove(i);

i--;

}

}

/\* Their should only be one root node left after this process, if the expression is balanced \*/

if(nodes.size()>1){

throw new Exception("Expression is not valid");

}

/\* Return the tree \*/

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

tree.root = nodes.get(0);

return tree;

}

/\*\*

\* Using a combination of looping and recursion this method creates a List of String tokens and creates a binary expression tree from them

\* @param expression The expression to be evaluated

\* @return A LinkedBinaryTree representation of the expression

\* @throws Exception If the expression is invalid or unbalanced

\*/

public static LinkedBinaryTree<String> makeTree(String expression) throws Exception{

ArrayList<String> tokens = new ArrayList<>();

String[] sTokens = expression.split(" ");

for(int i=0; i<sTokens.length; i++){

tokens.add(sTokens[i]);

}

return makeTree(tokens);

}

}

LinkedBinaryTree:

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

package Tree;

/\*\*

\*

\* @author Jacob Huesman

\*/

public class LinkedBinaryTree<E> extends AbstractBinaryTree<E> {

/\* Nested Node Class \*/

protected static class Node<E> implements Position<E> {

private E element;

private Node<E> parent;

private Node<E> left;

private Node<E> right;

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 of Nested Node Class \*/

/\*\*

\* Factory function to create a new node storing element e

\* @param e element e to be stored by node

\* @param parent parent node

\* @param left left child node

\* @param right right child node

\* @return new node for storing element e

\*/

protected Node<E> createNode(E e, Node<E> parent, Node<E> left, Node<E> right){

return new Node<>(e, parent, left, right);

}

//LinkedBinaryTree instance variables

protected Node<E> root = null;

private int size = 0;

/\*\*

\* Default constructor that constructs an empty binary tree

\*/

public LinkedBinaryTree(){}

/\*\*

\* Non-public utility that validates that the position is an instance of the nested Node class and returns it as a node

\* @param p position to validate

\* @return p casted as a node

\* @throws IllegalArgumentException if p is not an instance of Node or is no longer in the tree

\*/

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;

if(node.getParent() == node){

throw new IllegalArgumentException("p is no longer in the tree");

}

return node;

}

/\* Accessor Methods

/\*\*

\* Returns the size of the tree

\* @return size of tree

\*/

@Override

public int size(){

return size;

}

/\*\*

\* Returns the root Position of the tree (or null if tree is empty)

\* @return root Position of the tree

\*/

@Override

public Position<E> root(){

return root;

}

/\*\*

\* Returns the position of the node's parent

\* @param p position object of the node

\* @return position object of the parent to the node

\* @throws IllegalArgumentException

\*/

@Override

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

Node<E> node = validate(p);

return node.getParent();

}

/\*\*

\* Returns the positions left child (or null if no child exists)

\* @param p A position within the tree

\* @return the Positions of the left child (or null if no child exists)

\* @throws IllegalArgumentException if p is not a valid Position for this tree

\*/

@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)

\* @param p A position within the tree

\* @return the Position of the right child (or null if no child exists)

\* @throws IllegalArgumentException if p is not a valid Position for this tree

\*/

@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

\* @param e element

\* @return Position of the root of the tree containing element e

\* @throws IllegalStateException if the tree is not empty

\*/

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

\* @param p Position p in the tree

\* @param e element to be stored

\* @return Position of the left child of Position p

\* @throws IllegalArgumentException if p already has a left child

\*/

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

\* @param p Position p in the tree

\* @param e element to be stored

\* @return Position of the right child of Position p

\* @throws IllegalArgumentException if p already has a right child

\*/

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

\* @param p Position p to replace element at

\* @param e element to place in p

\* @return the Position of the element was stored in

\* @throws IllegalArgumentException

\*/

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

\* @param p a leaf of the tree

\* @param t1 an independent tree whose structure becomes the left child of p

\* @param t2 an independent tree whose structure becomes the right child of p

\* @throws IllegalArgumentException

\*/

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.

\* @param p Node to be removed

\* @return the Element held in Position p

\* @throws IllegalArgumentException if p has two children

\*/

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());

}

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(null);

return temp;

}

}

LinkedPositionalList:

/\*

\* Copyright 2014, Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser

\*

\* Developed for use with the book:

\*

\* Data Structures and Algorithms in Java, Sixth Edition

\* Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser

\* John Wiley & Sons, 2014

\*

\* This program is free software: you can redistribute it and/or modify

\* it under the terms of the GNU General Public License as published by

\* the Free Software Foundation, either version 3 of the License, or

\* (at your option) any later version.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\* You should have received a copy of the GNU General Public License

\* along with this program. If not, see <http://www.gnu.org/licenses/>.

\*/

package Tree;

import java.util.Iterator;

import java.util.NoSuchElementException;

/\*\*

\* Implementation of a positional list stored as a doubly linked list.

\*

\* @author Michael T. Goodrich

\* @author Roberto Tamassia

\* @author Michael H. Goldwasser

\*/

public class LinkedPositionalList<E> implements PositionalList<E> {

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

/\*\*

\* Node of a doubly linked list, which stores a reference to its

\* element and to both the previous and next node in the list.

\*/

private static class Node<E> implements Position<E> {

/\*\* The element stored at this node \*/

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

/\*\* A reference to the preceding node in the list \*/

private Node<E> prev; // reference to the previous node in the list

/\*\* A reference to the subsequent node in the list \*/

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

/\*\*

\* Creates a node with the given element and next node.

\*

\* @param e the element to be stored

\* @param p reference to a node that should precede the new node

\* @param n reference to a node that should follow the new node

\*/

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

element = e;

prev = p;

next = n;

}

// public accessor methods

/\*\*

\* Returns the element stored at the node.

\* @return the stored element

\* @throws IllegalStateException if node not currently linked to others

\*/

public E getElement() throws IllegalStateException {

if (next == null) // convention for defunct node

throw new IllegalStateException("Position no longer valid");

return element;

}

/\*\*

\* Returns the node that precedes this one (or null if no such node).

\* @return the preceding node

\*/

public Node<E> getPrev() {

return prev;

}

/\*\*

\* Returns the node that follows this one (or null if no such node).

\* @return the following node

\*/

public Node<E> getNext() {

return next;

}

// Update methods

/\*\*

\* Sets the node's element to the given element e.

\* @param e the node's new element

\*/

public void setElement(E e) {

element = e;

}

/\*\*

\* Sets the node's previous reference to point to Node n.

\* @param p the node that should precede this one

\*/

public void setPrev(Node<E> p) {

prev = p;

}

/\*\*

\* Sets the node's next reference to point to Node n.

\* @param n the node that should follow this one

\*/

public void setNext(Node<E> n) {

next = n;

}

} //----------- end of nested Node class -----------

// instance variables of the LinkedPositionalList

/\*\* Sentinel node at the beginning of the list \*/

private Node<E> header; // header sentinel

/\*\* Sentinel node at the end of the list \*/

private Node<E> trailer; // trailer sentinel

/\*\* Number of elements in the list (not including sentinels) \*/

private int size = 0; // number of elements in the list

/\*\* Constructs a new empty list. \*/

public LinkedPositionalList() {

header = new Node<>(null, null, null); // create header

trailer = new Node<>(null, header, null); // trailer is preceded by header

header.setNext(trailer); // header is followed by trailer

}

// private utilities

/\*\*

\* Verifies that a Position belongs to the appropriate class, and is

\* not one that has been previously removed. Note that our current

\* implementation does not actually verify that the position belongs

\* to this particular list instance.

\*

\* @param p a Position (that should belong to this list)

\* @return the underlying Node instance at that position

\* @throws IllegalArgumentException if an invalid position is detected

\*/

private Node<E> validate(Position<E> p) throws IllegalArgumentException {

if (!(p instanceof Node)) throw new IllegalArgumentException("Invalid p");

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

if (node.getNext() == null) // convention for defunct node

throw new IllegalArgumentException("p is no longer in the list");

return node;

}

/\*\*

\* Returns the given node as a Position, unless it is a sentinel, in which case

\* null is returned (so as not to expose the sentinels to the user).

\*/

private Position<E> position(Node<E> node) {

if (node == header || node == trailer)

return null; // do not expose user to the sentinels

return node;

}

// public accessor methods

/\*\*

\* Returns the number of elements in the list.

\* @return number of elements in the list

\*/

@Override

public int size() { return size; }

/\*\*

\* Tests whether the list is empty.

\* @return true if the list is empty, false otherwise

\*/

@Override

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

/\*\*

\* Returns the first Position in the list.

\*

\* @return the first Position in the list (or null, if empty)

\*/

@Override

public Position<E> first() {

return position(header.getNext());

}

/\*\*

\* Returns the last Position in the list.

\*

\* @return the last Position in the list (or null, if empty)

\*/

@Override

public Position<E> last() {

return position(trailer.getPrev());

}

/\*\*

\* Returns the Position immediately before Position p.

\* @param p a Position of the list

\* @return the Position of the preceding element (or null, if p is first)

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

@Override

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

Node<E> node = validate(p);

return position(node.getPrev());

}

/\*\*

\* Returns the Position immediately after Position p.

\* @param p a Position of the list

\* @return the Position of the following element (or null, if p is last)

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

@Override

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

Node<E> node = validate(p);

return position(node.getNext());

}

// private utilities

/\*\*

\* Adds an element to the linked list between the given nodes.

\* The given predecessor and successor should be neighboring each

\* other prior to the call.

\*

\* @param pred node just before the location where the new element is inserted

\* @param succ node just after the location where the new element is inserted

\* @return the new element's node

\*/

private Position<E> addBetween(E e, Node<E> pred, Node<E> succ) {

Node<E> newest = new Node<>(e, pred, succ); // create and link a new node

pred.setNext(newest);

succ.setPrev(newest);

size++;

return newest;

}

// public update methods

/\*\*

\* Inserts an element at the front of the list.

\*

\* @param e the new element

\* @return the Position representing the location of the new element

\*/

@Override

public Position<E> addFirst(E e) {

return addBetween(e, header, header.getNext()); // just after the header

}

/\*\*

\* Inserts an element at the back of the list.

\*

\* @param e the new element

\* @return the Position representing the location of the new element

\*/

@Override

public Position<E> addLast(E e) {

return addBetween(e, trailer.getPrev(), trailer); // just before the trailer

}

/\*\*

\* Inserts an element immediately before the given Position.

\*

\* @param p the Position before which the insertion takes place

\* @param e the new element

\* @return the Position representing the location of the new element

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

@Override

public Position<E> addBefore(Position<E> p, E e)

throws IllegalArgumentException {

Node<E> node = validate(p);

return addBetween(e, node.getPrev(), node);

}

/\*\*

\* Inserts an element immediately after the given Position.

\*

\* @param p the Position after which the insertion takes place

\* @param e the new element

\* @return the Position representing the location of the new element

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

@Override

public Position<E> addAfter(Position<E> p, E e)

throws IllegalArgumentException {

Node<E> node = validate(p);

return addBetween(e, node, node.getNext());

}

/\*\*

\* Replaces the element stored at the given Position and returns the replaced element.

\*

\* @param p the Position of the element to be replaced

\* @param e the new element

\* @return the replaced element

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

@Override

public E set(Position<E> p, E e) throws IllegalArgumentException {

Node<E> node = validate(p);

E answer = node.getElement();

node.setElement(e);

return answer;

}

/\*\*

\* Removes the element stored at the given Position and returns it.

\* The given position is invalidated as a result.

\*

\* @param p the Position of the element to be removed

\* @return the removed element

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

@Override

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

Node<E> node = validate(p);

Node<E> predecessor = node.getPrev();

Node<E> successor = node.getNext();

predecessor.setNext(successor);

successor.setPrev(predecessor);

size--;

E answer = node.getElement();

node.setElement(null); // help with garbage collection

node.setNext(null); // and convention for defunct node

node.setPrev(null);

return answer;

}

// support for iterating either positions and elements

//---------------- nested PositionIterator class ----------------

/\*\*

\* A (nonstatic) inner class. Note well that each instance

\* contains an implicit reference to the containing list,

\* allowing us to call the list's methods directly.

\*/

private class PositionIterator implements Iterator<Position<E>> {

/\*\* A Position of the containing list, initialized to the first position. \*/

private Position<E> cursor = first(); // position of the next element to report

/\*\* A Position of the most recent element reported (if any). \*/

private Position<E> recent = null; // position of last reported element

/\*\*

\* Tests whether the iterator has a next object.

\* @return true if there are further objects, false otherwise

\*/

public boolean hasNext() { return (cursor != null); }

/\*\*

\* Returns the next position in the iterator.

\*

\* @return next position

\* @throws NoSuchElementException if there are no further elements

\*/

public Position<E> next() throws NoSuchElementException {

if (cursor == null) throw new NoSuchElementException("nothing left");

recent = cursor; // element at this position might later be removed

cursor = after(cursor);

return recent;

}

/\*\*

\* Removes the element returned by most recent call to next.

\* @throws IllegalStateException if next has not yet been called

\* @throws IllegalStateException if remove was already called since recent next

\*/

public void remove() throws IllegalStateException {

if (recent == null) throw new IllegalStateException("nothing to remove");

LinkedPositionalList.this.remove(recent); // remove from outer list

recent = null; // do not allow remove again until next is called

}

} //------------ end of nested PositionIterator class ------------

//---------------- nested PositionIterable class ----------------

private class PositionIterable implements Iterable<Position<E>> {

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

} //------------ end of nested PositionIterable class ------------

/\*\*

\* Returns an iterable representation of the list's positions.

\* @return iterable representation of the list's positions

\*/

@Override

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

return new PositionIterable(); // create a new instance of the inner class

}

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

/\* This class adapts the iteration produced by positions() to return elements. \*/

private class ElementIterator implements Iterator<E> {

Iterator<Position<E>> posIterator = new PositionIterator();

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 list.

\* @return iterator of the list's elements

\*/

@Override

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

// Debugging code

/\*\*

\* Produces a string representation of the contents of the list.

\* This exists for debugging purposes only.

\*/

public String toString() {

StringBuilder sb = new StringBuilder("(");

Node<E> walk = header.getNext();

while (walk != trailer) {

sb.append(walk.getElement());

walk = walk.getNext();

if (walk != trailer)

sb.append(", ");

}

sb.append(")");

return sb.toString();

}

}

Positon:

/\*

\* Copyright 2014, Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser

\*

\* Developed for use with the book:

\*

\* Data Structures and Algorithms in Java, Sixth Edition

\* Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser

\* John Wiley & Sons, 2014

\*

\* This program is free software: you can redistribute it and/or modify

\* it under the terms of the GNU General Public License as published by

\* the Free Software Foundation, either version 3 of the License, or

\* (at your option) any later version.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\* You should have received a copy of the GNU General Public License

\* along with this program. If not, see <http://www.gnu.org/licenses/>.

\*/

package Tree;

/\*\*

\* An interface for a position which is an abstraction for the

\* location at which a single element is stored in a positional

\* container.

\*

\* @author Michael T. Goodrich

\* @author Roberto Tamassia

\* @author Michael H. Goldwasser

\*/

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;

}

PositionalList:

/\*

\* Copyright 2014, Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser

\*

\* Developed for use with the book:

\*

\* Data Structures and Algorithms in Java, Sixth Edition

\* Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser

\* John Wiley & Sons, 2014

\*

\* This program is free software: you can redistribute it and/or modify

\* it under the terms of the GNU General Public License as published by

\* the Free Software Foundation, either version 3 of the License, or

\* (at your option) any later version.

\*

\* This program is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU General Public License for more details.

\*

\* You should have received a copy of the GNU General Public License

\* along with this program. If not, see <http://www.gnu.org/licenses/>.

\*/

package Tree;

import java.util.Iterator;

/\*\*

\* An interface for positional lists.

\* @author Michael T. Goodrich

\* @author Roberto Tamassia

\* @author Michael H. Goldwasser

\* @see Position

\*/

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

/\*\*

\* Returns the number of elements in the list.

\* @return number of elements in the list

\*/

int size();

/\*\*

\* Tests whether the list is empty.

\* @return true if the list is empty, false otherwise

\*/

boolean isEmpty();

/\*\*

\* Returns the first Position in the list.

\*

\* @return the first Position in the list (or null, if empty)

\*/

Position<E> first();

/\*\*

\* Returns the last Position in the list.

\*

\* @return the last Position in the list (or null, if empty)

\*/

Position<E> last();

/\*\*

\* Returns the Position immediately before Position p.

\* @param p a Position of the list

\* @return the Position of the preceding element (or null, if p is first)

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

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

/\*\*

\* Returns the Position immediately after Position p.

\* @param p a Position of the list

\* @return the Position of the following element (or null, if p is last)

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

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

/\*\*

\* Inserts an element at the front of the list.

\*

\* @param e the new element

\* @return the Position representing the location of the new element

\*/

Position<E> addFirst(E e);

/\*\*

\* Inserts an element at the back of the list.

\*

\* @param e the new element

\* @return the Position representing the location of the new element

\*/

Position<E> addLast(E e);

/\*\*

\* Inserts an element immediately before the given Position.

\*

\* @param p the Position before which the insertion takes place

\* @param e the new element

\* @return the Position representing the location of the new element

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

Position<E> addBefore(Position<E> p, E e)

throws IllegalArgumentException;

/\*\*

\* Inserts an element immediately after the given Position.

\*

\* @param p the Position after which the insertion takes place

\* @param e the new element

\* @return the Position representing the location of the new element

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

Position<E> addAfter(Position<E> p, E e)

throws IllegalArgumentException;

/\*\*

\* Replaces the element stored at the given Position and returns the replaced element.

\*

\* @param p the Position of the element to be replaced

\* @param e the new element

\* @return the replaced element

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

E set(Position<E> p, E e) throws IllegalArgumentException;

/\*\*

\* Removes the element stored at the given Position and returns it.

\* The given position is invalidated as a result.

\*

\* @param p the Position of the element to be removed

\* @return the removed element

\* @throws IllegalArgumentException if p is not a valid position for this list

\*/

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

/\*\*

\* Returns an iterator of the elements stored in the list.

\* @return iterator of the list's elements

\*/

Iterator<E> iterator();

/\*\*

\* Returns the positions of the list in iterable form from first to last.

\* @return iterable collection of the list's positions

\*/

Iterable<Position<E>> positions();

}

Tree:

package Tree;

import java.util.Iterator;

/\*

\* To change this license header, choose License Headers in Project Properties.

\* To change this template file, choose Tools | Templates

\* and open the template in the editor.

\*/

/\*\*

\* Tree class outlined in Data Structures & Algorithms by Michael T Goodrich, Roberto Tamassia, and Michael H Goldwasser.

\* @author Jacob Huesman

\* @param <E>

\*/

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

/\*\*

\* Returns position of the root node

\* @return Position of root node

\*/

Position<E> root();

/\*\*

\* Returns the position of the node's parent

\* @param p position object of the node

\* @return position object of the parent to the node

\* @throws IllegalArgumentException

\*/

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

/\*\*

\* Returns an iterator containing the positions of the children of the node

\* @param p the node

\* @return iterator containing positions of the node's children

\* @throws IllegalArgumentException

\*/

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

/\*\*

\* Returns the number of children of a given node

\* @param p the node

\* @return number of children of the node

\* @throws IllegalArgumentException

\*/

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

/\*\*

\* Returns if the node is Internal

\* @param p the position of the node

\* @return true - if the node is internal; false - otherwise

\* @throws IllegalArgumentException

\*/

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

/\*\*

\* Returns if the node is External

\* @param p the position of the node

\* @return true - if the node is external; false - otherwise

\* @throws IllegalArgumentException

\*/

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

/\*\*

\* Returns if the node is the root of the tree

\* @param p the position of the node

\* @return true - if the node is the root; false - otherwise

\* @throws IllegalArgumentException

\*/

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

/\*\*

\* Returns the size of the tree

\* @return size of the tree

\*/

int size();

/\*\*

\* Returns if the tree is empty

\* @return true - if the tree is empty; false - otherwise

\*/

boolean isEmpty();

/\*\*

\* Returns an iterator of the elements stored in the tree

\* @return iterator of tree's elements

\*/

@Override

Iterator<E> iterator();

/\*\*

\* Returns an iterable collection of the positions of the tree

\* @return iterable collection of the tree's positions

\*/

Iterable<Position<E>> positions();

}







