**Note to grader:**

**The old code are in regular font and color**

**The codes that are critical for this implementation are shown in purple.**

**The new code/class are shown in olive green. Skip to the End to see.**

**Abstrack Binary Class(Same as last Lab)**

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;

}

}

**Abstract Tree Class(Old code)**

/\*\*

\*

\* @author Goodrich, Tamassia, Goldwasser

\*/

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

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

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

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

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

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

public Iterable<Position<E>> positions(){return preorder();}

/\*\*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 preoder, we add positon p before exploring subtress

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

preorderSubtree(c,snapshot);

}

public Iterable<Position<E>> preorder()

{

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

if(!isEmpty())

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

return snapshot;

}

//--------------------Posorder traversal----------------

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

private void postorderSubtree(Position<E> p, List<Position<E>> snapshot)

{

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

postorderSubtree(c,snapshot);

snapshot.add(p); //for postorder, we add position p after exploring subtrees

}

/\*\* Returns an iterable collection of positions of the tree, reported in postorder. \*/

public Iterable<Position<E>> postorder()

{

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

if(!isEmpty())

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

return snapshot;

}

//-------------------End of code for postOrder ------------------

//--------------- Nested iterarator class -----------

/\* 8.16 This class adapts the iteration produced by positions() tor return elements \*/

private class ElementIterator implements Iterator<E>

{

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

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

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

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

}

/\*\*\*\*\*\*\*\* End of Nested ElementIterator Class \*\*\*\*\*/

/\*\*returns an iterator of the elements stored in the tree \*/

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

/\*\*

\* Returns an iterable collection of positions of the tree in breadth-first order.

\* code fragment 8.21

\*/

public Iterable<Position<E>> breathFirst()

{

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

if(!isEmpty())

{

Queue<Position<E>> fringe = new LinkedQueue<>();

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;

}

}

**BinaryTree Interface(Old Code)**

/\*\*

\* 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(Old code)**

/\*\*

\*

\* @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;

}

}

**SinglyLinkedList Class(old code)**

/\*\*

\*

\* @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;

}

}

**LinkedStack Class (Old Code)**

/\*\*

\* This is based on textbook code fragment 6.4

\* This stack implementation add the last in element at the head.

\* @author Richelin Metellus

\* @param <E>

\*/

public class LinkedStack<E> implements Stack<E> {

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

public LinkedStack(){} //empty list

@Override

public int size()

{

return list.size();

}

@Override

public boolean isEmpty()

{

return list.isEmpty();

}

@Override

public void push(E element)

{

list.addFirst(element);

}

@Override

public E top(){return list.first();}

@Override

public E pop(){return list.removeFirst();}

}

**Positional List Interface(Old Code)**

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragement 7.8

\*/

/\*\*

\* An interface for positional lists.

\*/

public interface PositionalList<E> {

/\*\*

\* @return the number of elements in the list.

\*/

int size( );

/\*\*

\* @return true if the list is empty.

\*/

boolean isEmpty( );

/\*\*

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

\*/

Position<E> first( );

/\*\*

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

\*/

Position<E> last( );

/\*\*

\* @param p a position in the list,

\* @return position immediately before p ( or null if p is first ).

\* @throws IllegalArgumentException if p is not in list.

\*/

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

/\*\*

\* @param p a position in the list,

\* @return position immediately after p ( or null if p is last ).

\* @throws IllegalArgumentException if p is not in list.

\*/

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

/\*\*

\* @param e element to be inserted at front of list

\* @return position of inserted element

\*/

Position<E> addFirst( E e );

/\*\*

\* @param e element to be inserted at back of list

\* @return position of inserted element

\*/

Position<E> addLast( E e );

/\*\*

\* @param p position to be inserted before

\* @param e element to be inserted before position p

\* @return position of e

\* @throws IllegalArgumentException if p not in list

\*/

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

/\*\*

\* @param p position to be inserted after

\* @param e element to be inserted after position p

\* @return position of e

\* @throws IllegalArgumentException if p not in list

\*/

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

/\*\*

\* @param p position to store element at

\* @param e element to be stored at p

\* @return the element that is replaced

\* @throws IllegalArgumentException if p is not in list

\*/

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

/\*\*

\* @param p position of element to be removed

\* @return removed element

\* @throws IllegalArgumentException if p not in list

\*/

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

}

**LinkedPostionalList Class**

import java.util.Iterator;

import java.util.NoSuchElementException;

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragements 7.9, 7.10, 7.11, 7.12 & 7.14

\*

\* toString method added by Latimer

\*/

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

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

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

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

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

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

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

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

element = e;

prev = p;

next = n;

}

@Override

public E getElement( ) throws IllegalStateException

{

if ( next == null )

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

return element;

}

public Node<E> getPrev( )

{

return prev;

}

public Node<E> getNext( )

{

return next;

}

public void setElemetn( E e )

{

element = e;

}

public void setPrev( Node<E> p )

{

prev = p;

}

public void setNext( Node<E> n )

{

next = n;

}

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

/\*\*

\* Data Structures & Algorithms 6th Edition

\* Goodrick, Tamassia, Goldwasser

\* Code Fragement 7.14

\*/

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

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

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

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

/\*\* Tests whether the iterator has a next object. \*/

@Override

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

/\*\* Returns the next position in the iterator. \*/

@Override

public Position<E> next( ) throws NoSuchElementException {

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

recent = cursor;

cursor = after( cursor );

return recent;

}

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

@Override

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>>{

@Override

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

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

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

\* @return \*/

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( );

@Override

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

@Override

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

@Override

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

}

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

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

// instance variables of the LinkedPositionalList

private Node<E> header; // header sentinel

private Node<E> trailer; // trailer sentinel

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

public LinkedPositionalList( ){

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

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

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

}

// private utilities

/\*\*

\* @param p position to validate

\* @return node if position is valid

\* @throws IllegalArgumentException if p no longer in list or p is not a position

\*/

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 )

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

return node;

}

/\*\*

\* @param node to be returned as position if not header or trailer

\* @return position of node

\*/

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

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

return null;

return node;

}

// public accessor methods

/\*\*

\* @return number of elements in linked list

\*/

@Override

public int size( ){

return size;

}

/\*\*

\* @return true if list is empty, false other wise

\*/

@Override

public boolean isEmpty( ){

return ( size == 0 );

}

/\*\*

\* @return the first position in linked list (null if empty).

\*/

@Override

public Position<E> first( ){

return position( header.getNext( ) );

}

/\*\*

\* @return the last position in linked list (null if empty).

\*/

@Override

public Position<E> last( ){

return position( trailer.getPrev( ) );

}

/\*\*

\* @param p position to get position immediately before

\* @return position before p

\* @throws IllegalArgumentException if p not valid

\*/

@Override

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

Node<E> node = validate( p );

return position( node.getPrev( ) );

}

/\*\*

\* @param p position to get immediately after

\* @return position after p

\* @throws IllegalArgumentException if p not valid

\*/

@Override

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

Node<E> node = validate( p );

return position( node.getNext( ) );

}

// private utilities

/\*\*

\* @param e element to be added

\* @param pred node to add element after

\* @param succ node to add element before

\* @return position of newly added element

\*/

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

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

pred.setNext(newest);

succ.setPrev(newest);

size++;

return newest;

}

// public update methods

/\*\*

\* @param e element to be added just after header

\* @return position of newly added element

\*/

@Override

public Position<E> addFirst(E e) {

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

}

/\*\*

\* @param e element to be added just before trailer

\* @return position of newly added element

\*/

@Override

public Position<E> addLast( E e ){

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

}

/\*\*

\*

\* @param p position to add element before

\* @param e element to be added

\* @return position of newly added element

\* @throws IllegalArgumentException if p is not valid

\*/

@Override

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

Node<E> node = validate( p );

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

}

/\*\*

\* @param p position to add element after

\* @param e element to be added

\* @return position of newly added element

\* @throws IllegalArgumentException if p is not valid

\*/

@Override

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

Node<E> node = validate( p );

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

}

/\*\*

\* @param p position of node to update

\* @param e new element for node

\* @return old element in node before update

\* @throws IllegalArgumentException if p not valid

\*/

@Override

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

Node<E> node = validate( p );

E answer = node.getElement( );

node.setElemetn( e );

return answer;

}

/\*\*

\* @param p position to be removed

\* @return element that was removed

\* @throws IllegalArgumentException if p not valid

\*/

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.setElemetn( null );

node.setNext( null );

node.setPrev( null );

return answer;

}

}

**PositionalQueue Interface (New code)**

/\*\*

\*@version 03/26/2017

\* @author Richelin Metellus

\*/

public interface PositionalQueue<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

\* @return

\*/

Position<E> 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();

}

**LinkedPositionalQueue Class(New code)**

import java.util.Iterator;

import java.util.NoSuchElementException;

/\*\*

\* This is an implementation of a queue with each item in the

\* queue having a position that can be use to iterate the queue and

\* do some operations such as validate a mathematical expression represented

\* in infix notation of lab 09.

\*

\* @author Richelin Metellus

\* @version 03/26/2017

\*/

public class LinkedPositionalQueue<E> implements PositionalQueue<E> {

private LinkedPositionalList<E> list; // doubly linked list for a queue with positions

public LinkedPositionalQueue()

{

list = new LinkedPositionalList(); // new queu relies on the inital empty list

}

/\*\*

\*

\* @return the number of element in the queue

\*/

@Override

public int size() { return list.size();}

/\*\*

\*

\* @return true if the queue is empty

\*/

@Override

public boolean isEmpty() { return list.isEmpty();}

/\*\*

\*

\* @param e element to b inserted at the end of the queue

\* @return the position of the new element inserted

\*/

@Override

public Position<E> enqueue(E e)

{

Position<E> p ;

p =list.addLast(e);

return p;

}

/\*\*

\*

\* @return but does not remove the element at the front of the queue

\*/

@Override

public E first() { return list.first().getElement();}

/\*\*

\*

\* @return the element removed at the front of the queue

\*/

@Override

public E dequeue() {

return list.remove(list.first());

//(Position<E>) list.remove(list.last());

}

/\*\*

\*

\* @return does not remove the position of the item at the front of the queue

\*/

public Position<E> firstPosition()

{

return list.first();

}

/\*\*

\*

\* @return the position of the next item in the queue

\*/

public Position<E> positionAfterFirst()

{

return list.after(list.first());

}

//------------------------------Start of Position Iterator Class-------------

private class PositionIterator implements Iterator<Position<E>>

{

private Position<E> cursor = list.first();

private Position<E> recent = null;

@Override

public boolean hasNext()

{

return (cursor != null);

}

@Override

public Position<E> next() throws NoSuchElementException

{

if(cursor == null) throw new NoSuchElementException(" No More Position");

recent = cursor;

cursor = list.after(cursor);

return recent;

}

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

@Override

public void remove() throws IllegalStateException

{

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

list.remove(recent);

recent = null;

}

}

//-----------------PositionIterator End-----------

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*Nested PositionIterable Class\*\*\*\*\*\*

private class PositionIterable implements Iterable<Position<E>>

{

@Override

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

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*End of Nested PostionIterableClass\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\*\*

\*

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

\*/

public Iterable<Position<E>> positions()

{

return new PositionIterable(); // after this execute, think of cursor = position/address of first luckyNumber object.

}

/\*\*

\* This utility method is important as it retains the contents

\* of the original queue without having to dequeue the original one.

\*

\* @return a copy of a queue with each item having their own position.

\*/

public LinkedPositionalQueue<E> clone() {

LinkedPositionalQueue<E> temp = new LinkedPositionalQueue<>();

for (Position<E> e : list.positions()) {

if (e != null) {

temp.enqueue(e.getElement());

}

}

return temp;

}

}

**Shunting Yard Algorithm Client**

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Iterator;

import java.util.LinkedList;

import java.util.List;

import java.util.Scanner;

/\*\*

\* This algorithm allow the evaluation of mathematical expressions

\* @author Richelin Metellus

\* @version 03/27/2017

\*/

public class ShuntingYardClient {

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

LinkedPositionalQueue<String> infixQueue = new LinkedPositionalQueue();

LinkedPositionalQueue<String> postfixQueue = new LinkedPositionalQueue();

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

boolean valid = false;

Scanner scan = new Scanner(System.in);

String filePath;

String fileName;

File f = new File("C:\\");

while (!valid) {

System.out.print("Enter the filePath: ");

filePath = scan.nextLine();

System.out.print("Enter the fileName: ");

fileName = scan.nextLine();

f = new File(filePath,fileName);

if (f.exists()) {

valid = true;

}

else {

System.out.println("Invalid file path or name. Enter a valid path");

}

}

Scanner scanFile = new Scanner(f);

List<String> expressionList = new LinkedList<>();

while (scanFile.hasNextLine()) {

String inLine = scanFile.nextLine(); // take each line which equal an expression from the file

expressionList.add(inLine); //

}

int temp = expressionList.size();

String[] exp = new String[temp]; // to save each line of the file(which is an expression)

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

{

exp[i] = expressionList.get(i);

System.out.printf("\n------------------------------\n\n%s ",exp[i]);

infixQueue = stringToQueue(exp[i]);

if (validateExpression(infixQueue)) // boolean function to check for validity of the expression

{

System.out.println(" It is a valid expression");

postfixQueue = infixToPostfix(infixQueue);

System.out.printf(" Expression in Postfix:\t\t %s \n\n", postfixToString(postfixQueue));

expressionTree = createExpressionTree(postfixQueue);

//System.out.println("expressionTree root element" + expressionTree.root().getElement());

//PreOrder traversal

Iterator<Position<String>> preOrderIterator = expressionTree.preorder().iterator();

while (preOrderIterator.hasNext()) {

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

}

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

// Inorder

Iterator<Position<String>> inOrderIterator = expressionTree.inorder().iterator();

while (inOrderIterator.hasNext()) {

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

}

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

// PostOrder Traversal

Iterator<Position<String>> postOrderIterator = expressionTree.postorder().iterator();

while (postOrderIterator.hasNext()) {

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

}

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

System.out.printf(" Expression Result\t %10.3f \n", evaluatePostfix(postfixQueue));

}

else

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

}

}// End of main

//--------------------------Static methods-----------------------

/\*\*

\*

\* @param expression mathematical expression as string

\* @return a queue representative of the string expression

\*/

public static LinkedPositionalQueue<String> stringToQueue(String expression) {

LinkedPositionalQueue<String> infix = new LinkedPositionalQueue<>();

Scanner scan = new Scanner(expression);

scan.useDelimiter(" ");

while (scan.hasNext()) {

infix.enqueue(scan.next());

}

return infix; // infix is just as the expression in the file.

}

/\*\*

\*

\* @param infix a queue of the expression without the spaces

\* @return true if the parentheses are balanced and expression format is valid

\*/

public static boolean validateExpression(LinkedPositionalQueue<String> infix)

{

LinkedStack<String> pStack = new LinkedStack<>(); // stack for the symbol/parenthesis

LinkedPositionalQueue<String> infixClone = infix.clone();

Iterator<Position<String>> infixIter = infixClone.positions().iterator(); // iterator to loop through

String currentToken =""; //all the items each having a position in the queue

String previousToken = "";

String nextToken ="";

while(infixIter.hasNext())

{

previousToken = currentToken;

currentToken = infixIter.next().getElement();

if(infixClone.positionAfterFirst() != null)

nextToken = infixClone.positionAfterFirst().getElement();

if(isOperand(currentToken) && infixIter.hasNext())

{

infixClone.dequeue(); // allow my current to advance

if(isOpenParenthesis(nextToken) || isOperand(previousToken))

return false;

}

if(isOpenParenthesis(currentToken))

{

pStack.push(currentToken);

infixClone.dequeue();

if(isOperator(nextToken))

return false;

}

else if(isCloseParenthesis(currentToken))

{

if(pStack.isEmpty() || !areParenthesesPaired(pStack.top(),currentToken))

return false;

else

{

pStack.pop(); // i.e the parentheses match delete both

infixClone.dequeue();

}

}

else if(isOperator(currentToken))

{

if(!infixIter.hasNext()|| isOperator(nextToken))

return false; // will take care of this situation )+

else

infixClone.dequeue();

}

}

return true && pStack.isEmpty(); // if unbalanced parentheses pStack is not empty and will contain open symbols

}

/\*\*

\*

\* @param infix a queue of the expression in its original order without the spaces

\* @return the postfix form of the expression

\*/

public static LinkedPositionalQueue<String> infixToPostfix(LinkedPositionalQueue<String> infix) {

LinkedPositionalQueue<String> postfix = new LinkedPositionalQueue<>();

LinkedPositionalQueue<String> infixClone = infix.clone();

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

while (!infixClone.isEmpty())

{

if (isOperand(infixClone.first())) {

postfix.enqueue(infixClone.dequeue());

}

else if (isOperator(infixClone.first()))

{

while(!stack.isEmpty() && !isOpenParenthesis(stack.top())

&& hasHigherPrecedence(stack.top(),infixClone.first()))

{

postfix.enqueue(stack.pop()); // enque the element(s) of higher precedence in the stack.

} // till it finds an open parenthesis/symbol in the stack

stack.push(infixClone.dequeue()); // then push the operator on top of openingParenthese(s)

}

else if (isOpenParenthesis(infixClone.first()))

stack.push(infixClone.dequeue());

else if (isCloseParenthesis(infixClone.first()))

{

while(!stack.isEmpty() && !isOpenParenthesis(stack.top()))

postfix.enqueue(stack.pop());

infixClone.dequeue(); // to delete the close parenthesis on the infix

stack.pop(); // delete the open symbol/parenthesis in the stack

}

}

while(!stack.isEmpty()) // finally move all remaining element/operators in the stack to postfix queue

postfix.enqueue(stack.pop());

return postfix;

}

/\*\*

\*

\* @param postfix a queue with each element having a position

\* @return the value of the expression

\*/

// assuming the postfix is a valid expression

public static double evaluatePostfix(LinkedPositionalQueue<String> postfix)

{

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

LinkedPositionalQueue<String> postfixClone = postfix.clone();

double result;

while (!postfixClone.isEmpty())

{

if(isOperand(postfixClone.first()))

S.push(postfixClone.dequeue());

else if(isOperator(postfixClone.first()) && (S.size()>=2))

{

String op1 = S.pop();

String op2 = S.pop();

result = calculate(op2,op1, postfixClone.dequeue());

S.push(Double.toString(result));

}

}

return Double.parseDouble(S.pop());

}

/\*\*

\*

\* @param postfix a queue with each item having their own position

\* @return A binary tree of the expression

\*/

public static LinkedBinaryTree<String> createExpressionTree(LinkedPositionalQueue<String> postfix)

{

LinkedStack<LinkedBinaryTree> treeStack = new LinkedStack();

LinkedPositionalQueue<String>postfixClone = postfix.clone();

int postfixSize = postfixClone.size();

while(!postfixClone.isEmpty())

{

if(isOperand(postfixClone.first()))

{

LinkedBinaryTree<String> T1 = new LinkedBinaryTree();

T1.addRoot(postfixClone.dequeue());

treeStack.push(T1);

}

else if(isOperator(postfixClone.first()))

{

LinkedBinaryTree<String> expressionTree = new LinkedBinaryTree();

LinkedBinaryTree<String> rightChild = treeStack.pop();

LinkedBinaryTree<String> leftChild = treeStack.pop();

expressionTree.attach(expressionTree.addRoot(postfixClone.dequeue()), leftChild, rightChild);

treeStack.push(expressionTree);

//System.out.println("expressionTree root element" + expressionTree.root().getElement());

}

}

return treeStack.pop();

}

//--------------------- utility methods --------------------------

public static boolean isOperand(String S) {

try { Double.parseDouble(S);}

catch(NumberFormatException e){ return false;}

return true;

}

// this method assign a weight for each operator

// this method is critical in defining operator precedence

public static int getOperatorPriority(String S) {

int weight = -1;

switch (S) {

case "\*":

case "/":

weight = 2;

break;

case "+":

case "-":

weight = 1;

}

return weight;

}

// operator with higher precedence has higher priority weight

public static boolean hasHigherPrecedence(String operator1, String operator2) {

int priorityWeight1 = getOperatorPriority(operator1);

int priorityWeight2 = getOperatorPriority(operator2);

return priorityWeight1 >= priorityWeight2;

}

public static boolean isOperator(String S) {

return S.equals("+") ||S.equals("-") ||S.equals("\*")

||S.equals("/");

}

public static boolean isOpenParenthesis(String S) {

char symbol = S.charAt(0);

return (symbol == '(' || symbol == '[' || symbol == '{');

}

public static boolean isCloseParenthesis(String S) {

char symbol = S.charAt(0);

return (symbol == ')' || symbol == ']' || symbol == '}');

}

public static boolean areParenthesesPaired(String open, String close)

{

return (open.equals("(")&& close.equals(")"))

||(open.equals("{")&& close.equals("}"))

||(open.equals("[")&& close.equals("]"));

}

public static double calculate(String op1, String op2,String operator )

{

double num1 = Double.parseDouble(op1);

double num2 = Double.parseDouble(op2);

switch(operator){

case "+":

return num1+ num2;

case "-":

return num1 - num2;

case "\*":

return num1 \* num2;

case "/":

return num1 / num2;

}

return 0; // get rid of compiler whining

}

/\*\*

\*

\* @param postfix a queue in postfix form

\* @return a String representative of the postfix queue

\*/

public static String postfixToString(LinkedPositionalQueue<String> postfix)

{

String postfixString = "";

LinkedPositionalQueue<String> postfixClone = postfix.clone();

int queueSize = postfixClone.size();

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

postfixString += postfixClone.dequeue()+" ";

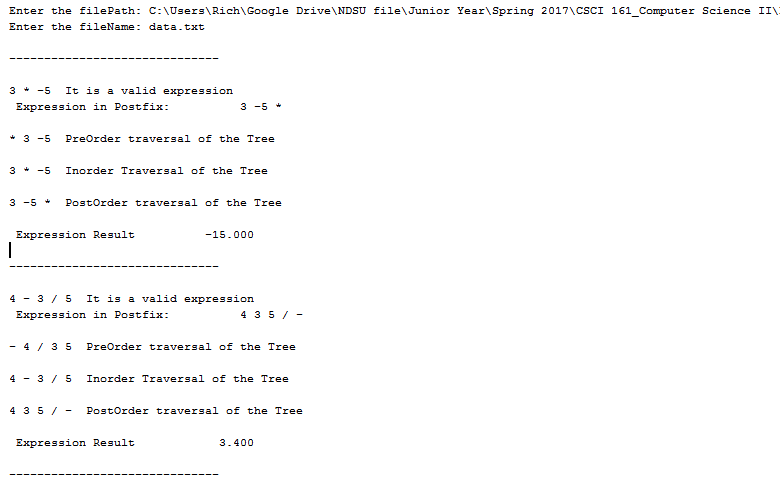
return postfixString;

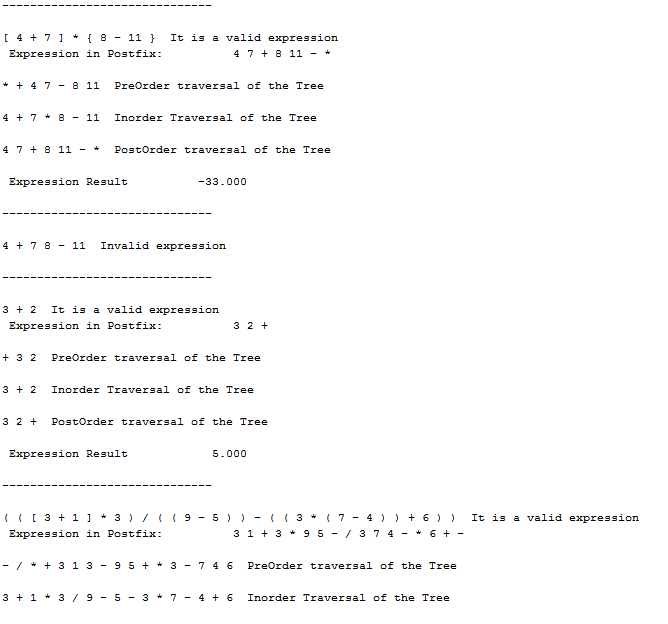
}

//------------------------------End of utility Methods------------------

}

**Snip of the program output**





**Complete output to Show all invalid expression it can handle**

------------------------------

3 \* -5 It is a valid expression

Expression in Postfix: 3 -5 \*

\* 3 -5 PreOrder traversal of the Tree

3 \* -5 Inorder Traversal of the Tree

3 -5 \* PostOrder traversal of the Tree

Expression Result -15.000

------------------------------

4 - 3 / 5 It is a valid expression

Expression in Postfix: 4 3 5 / -

- 4 / 3 5 PreOrder traversal of the Tree

4 - 3 / 5 Inorder Traversal of the Tree

4 3 5 / - PostOrder traversal of the Tree

Expression Result 3.400

------------------------------

( 4 - 3 ) / 5 It is a valid expression

Expression in Postfix: 4 3 - 5 /

/ - 4 3 5 PreOrder traversal of the Tree

4 - 3 / 5 Inorder Traversal of the Tree

4 3 - 5 / PostOrder traversal of the Tree

Expression Result 0.200

------------------------------

4 + ( 7 + 2 ) It is a valid expression

Expression in Postfix: 4 7 2 + +

+ 4 + 7 2 PreOrder traversal of the Tree

4 + 7 + 2 Inorder Traversal of the Tree

4 7 2 + + PostOrder traversal of the Tree

Expression Result 13.000

------------------------------

[ 4 + 7 ] \* { 8 - 11 } It is a valid expression

Expression in Postfix: 4 7 + 8 11 - \*

\* + 4 7 - 8 11 PreOrder traversal of the Tree

4 + 7 \* 8 - 11 Inorder Traversal of the Tree

4 7 + 8 11 - \* PostOrder traversal of the Tree

Expression Result -33.000

------------------------------

4 + 7 8 - 11 Invalid expression

------------------------------

3 + 2 It is a valid expression

Expression in Postfix: 3 2 +

+ 3 2 PreOrder traversal of the Tree

3 + 2 Inorder Traversal of the Tree

3 2 + PostOrder traversal of the Tree

Expression Result 5.000

------------------------------

( ( [ 3 + 1 ] \* 3 ) / ( ( 9 - 5 ) ) - ( ( 3 \* ( 7 - 4 ) ) + 6 ) ) It is a valid expression

Expression in Postfix: 3 1 + 3 \* 9 5 - / 3 7 4 - \* 6 + -

- / \* + 3 1 3 - 9 5 + \* 3 - 7 4 6 PreOrder traversal of the Tree

3 + 1 \* 3 / 9 - 5 - 3 \* 7 - 4 + 6 Inorder Traversal of the Tree

3 1 + 3 \* 9 5 - / 3 7 4 - \* 6 + - PostOrder traversal of the Tree

Expression Result -12.000

------------------------------

( ( 3 + 1 ) \* 3 / ( ( 9 - 5 ) ) - ( ( 3 \* ( 7 - 4 ) ) + 6 ) ) It is a valid expression

Expression in Postfix: 3 1 + 3 \* 9 5 - / 3 7 4 - \* 6 + -

- / \* + 3 1 3 - 9 5 + \* 3 - 7 4 6 PreOrder traversal of the Tree

3 + 1 \* 3 / 9 - 5 - 3 \* 7 - 4 + 6 Inorder Traversal of the Tree

3 1 + 3 \* 9 5 - / 3 7 4 - \* 6 + - PostOrder traversal of the Tree

Expression Result -12.000

------------------------------

3 + 1 \* 3 / 9 - 5 - 3 \* 7 - 4 + 6 It is a valid expression

Expression in Postfix: 3 1 3 \* 9 / + 5 - 3 7 \* - 4 - 6 +

+ - - - + 3 / \* 1 3 9 5 \* 3 7 4 6 PreOrder traversal of the Tree

3 + 1 \* 3 / 9 - 5 - 3 \* 7 - 4 + 6 Inorder Traversal of the Tree

3 1 3 \* 9 / + 5 - 3 7 \* - 4 - 6 + PostOrder traversal of the Tree

Expression Result -20.667

------------------------------

3 + 4 - Invalid expression

------------------------------

42 It is a valid expression

Expression in Postfix: 42

42 PreOrder traversal of the Tree

42 Inorder Traversal of the Tree

42 PostOrder traversal of the Tree

Expression Result 42.000

------------------------------

8 \* 24 / ( 4 + 3 Invalid expression

------------------------------

( ( ( 7 + 5 ) \* ( 9 - 3 ) ) / ( ( 15 + 24 ) + ( ( 6 - 1 ) - 5 ) ) \* 9 ) It is a valid expression

Expression in Postfix: 7 5 + 9 3 - \* 15 24 + 6 1 - 5 - + / 9 \*

\* / \* + 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 Inorder Traversal of the Tree

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

Expression Result 16.615

------------------------------

7 + - 3 Invalid expression

------------------------------

( ( 2 \* ( 7 - 1 ) ) + ( 3 \* 9 ) ) It is a valid expression

Expression in Postfix: 2 7 1 - \* 3 9 \* +

+ \* 2 - 7 1 \* 3 9 PreOrder traversal of the Tree

2 \* 7 - 1 + 3 \* 9 Inorder Traversal of the Tree

2 7 1 - \* 3 9 \* + PostOrder traversal of the Tree

Expression Result 39.000

------------------------------

() Invalid expression

BUILD SUCCESSFUL (total time: 18 seconds)