EECS 2011 Assignment 3

**High Performance, Robust Embedded Automotive Information System**

Date: December 5, 2017

Course: EECS 2011

|  |  |  |  |
| --- | --- | --- | --- |
| No | Name | Student ID | EECS ID |
| 1 | Jason Skinner | 215115678 | skinner1 |
| 2 | Sohrab Oryakhel | 214302145 | sohrab23 |

**Report**

Method one: **AVL TREE MAP**

We used the AVL tree because it has a speed of O(log n) for all methods of search, insert and delete. This allow the vehicle to create the error in the log very quickly allowing little or no lag time between the processes.

Difficulties:

There were many difficulties in the implementation of the AVL Tree. Some of the functions did not work as they were designed and had to be manipulated to compute in proper time.

ALL Testing:

The testing was done with samples instances that were created to verify that the log could insert, delete, and search in high speed. Instances are created without duplication based on the priority of the error and the priority of the part. Test were created at random with random errors and random parts. These were then inserted into the tree so that they could be read and searched later. We expected that the Mechanic with the Scanner would be the one that would do the deletions from the list as the part was fixed / repaired.

**Code**

package eecs2011a3;

import java.util.TreeMap;

public class A3Version1 {

public static final TreeMap<String, EntryValue> ITEMS\_LOGS = new TreeMap<String, EntryValue>();

public A3Version1() {

; // intentionally blank

}

public void run() {

}

public EntryValue getInstances(Parts part, Errors error) {

String priority = error.toString().substring(1, 3);

if (ITEMS\_LOGS.containsKey("" + priority + ";" + part.ordinal())) {

return ITEMS\_LOGS.get("" + priority + ";" + part.ordinal());

} else {

EntryValue ev = new EntryValue(part, error);

ITEMS\_LOGS.put("" + priority + ";" + part.ordinal(), ev);

Log.add("" + priority + ";" + part.ordinal(), ev);

return ITEMS\_LOGS.get("" + priority + ";" + part.ordinal());

}

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public static void main(String[] args) {

// TODO Auto-generated method stub

// TEST\_CASES.runTestsVersion1();

Log.retrieveLog();

TEST\_CASES.runTestsVersion1();

A3Version1 a3 = new A3Version1();

a3.getInstances(Parts.BACK\_RIGHT\_TIRE, Errors.P88\_HIGH\_PRESSURE);

}

}

package eecs2011a3;

import java.util.Comparator;

import java.util.Date;

import java.util.TreeMap;

@SuppressWarnings("all")

public class AVLTreeMap<K, V> {

Node root;

public AVLTreeMap() {

}

/\*

\* add the node to the root at the bottom left of the tree and rebalances the tree

\*/

public void insertNode(Node node, Node newNode, String key) {

if (root == null) {

root = new Node<String>(key);

} else {

if (key != null) {

if (height(node) > 1) {

insertNode(node.left, null, key);

} else {

node.left = new Node<String>(key);

}

} else {

Node bot = getBottomNode(node);

bot.left = newNode;

root = rightRotate(node);

}

}

}

/\* finds the bottom left node \*/

public Node getBottomNode(Node node) {

boolean done = false;

Node tempNode = node;

Node result = null;

while (!done) {

if (node.left == null) {

result = tempNode;

done = true;

} else {

tempNode = tempNode.left;

}

}

return result;

}

/\*

\* return the height of the node given.

\*/

protected int height(Node node) {

return node.height;

}

private int max(int a, int b) {

return (a > b) ? a : b;

}

/\* performs a rotate Right based on the AVL algorithm \*/

private Node rightRotate(Node y) {

Node x = y.left;

Node T2 = x.right;

// Perform rotation

x.right = y;

y.left = T2;

// Update heights

y.height = max(height(y.left), height(y.right)) + 1;

x.height = max(height(x.left), height(x.right)) + 1;

// Return new root

return x;

}

/\* performs a rotate left based on the AVL algorithm \*/

private Node leftRotate(Node x) {

Node y = x.right;

Node T2 = y.left;

// Perform rotation

y.left = x;

x.right = T2;

// Update heights

x.height = max(height(x.left), height(x.right)) + 1;

y.height = max(height(y.left), height(y.right)) + 1;

// Return new root

return y;

}

/\*checks to see if the tree is balanced within one degree \*/

protected boolean isBalanced(Node p) {

return Math.abs(height(p.left) - height(p.right)) <= 1;

}

/\* removes the selected node and re-balances the remainder \*/

private void delete(Node node) {

Node oldRight = node.right;

node = node.left;

insertNode(node, oldRight, null);

}

}

package eecs2011a3;

import java.text.DateFormat;

import java.text.ParseException;

import java.text.SimpleDateFormat;

import java.util.Date;

public class EntryValue {

Parts sensor;

int priority;

Errors error;

Date date;

public EntryValue(Parts sensor, Errors error) {

this.sensor = sensor;

this.error = error;

this.date = new Date();

setPriority();

}

public EntryValue(String fromLog) {

System.out.println(fromLog.substring(fromLog.indexOf(" ") + 1, fromLog.indexOf(",")));

this.sensor = Parts.valueOf(fromLog.substring(fromLog.indexOf(" ") + 1, fromLog.indexOf(",")));

fromLog = fromLog.substring(fromLog.indexOf(",") + 1);

System.out.println(fromLog.substring(fromLog.indexOf(" ") + 1, fromLog.indexOf(",")));

this.error = Errors.valueOf(fromLog.substring(fromLog.indexOf(" ") + 1, fromLog.indexOf(",")));

fromLog = fromLog.substring(fromLog.indexOf(",") + 1);

DateFormat format = new SimpleDateFormat("MMM dd HH:mm:ss yyyy");

try {

System.out.println(fromLog.substring(11, 26) + " " + fromLog.substring(31));

this.date = format.parse(fromLog.substring(11, 26) + " " + fromLog.substring(31));

} catch (ParseException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

setPriority();

}

public void setPriority() {

this.priority = Integer.parseInt(error.toString().substring(1, 2));

}

public String toString() {

return "Part: " + sensor.toString() + ",Error: " + error.toString() + ", Date: " + date;

}

}

package eecs2011a3;

public enum Errors {

P98\_UNKNOWN, P99\_LOW\_PRESSURE, P88\_HIGH\_PRESSURE, P99\_LOW\_BATTERY, B01\_NO\_BRAKES,

}

package eecs2011a3;

import java.util.Date;

import java.util.Map;

import java.util.TreeMap;

/\* Each item has its own log so a mechanic can easily find a log based on an item

\*

\*

\*/

public class Items {

int priority;

Parts item;

TreeMap<Date, String> Log = new TreeMap<Date, String>();

public Items(Parts item, String error) {

this.item = item;

addLog(error);

}

private void addLog(String Error) {

Log.put(new Date(), Error); //Error has priority built into it ("pp, Error")

}

private void adjustPriority() {

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

if (priority < Integer.parseInt(Log.get(i).substring(0,2))) {

priority = Integer.parseInt(Log.get(i).substring(0,2)); //sets highest priority found to this.priority

}

}

}

}

package eecs2011a3;

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.io.PrintWriter;

import java.util.TreeMap;

public class Log {

private static final TreeMap<String, EntryValue> ITEMS\_LOGS\_BACKUP = new TreeMap<String, EntryValue>();

private static final String FILENAME = "log.txt";

public static void saveLog() {

try {

// Writes to a text file.

PrintWriter out = new PrintWriter(FILENAME);

int size = A3Version1.ITEMS\_LOGS.size();

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

out.println(A3Version1.ITEMS\_LOGS.firstEntry());

System.out.println("OUT: " + A3Version1.ITEMS\_LOGS.firstEntry());

ITEMS\_LOGS\_BACKUP.put("" + A3Version1.ITEMS\_LOGS.firstKey(),

A3Version1.ITEMS\_LOGS.get(A3Version1.ITEMS\_LOGS.firstKey()));

A3Version1.ITEMS\_LOGS.remove("" + A3Version1.ITEMS\_LOGS.firstKey());

}

out.close();

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

A3Version1.ITEMS\_LOGS.put("" + ITEMS\_LOGS\_BACKUP.firstKey(),

ITEMS\_LOGS\_BACKUP.get(ITEMS\_LOGS\_BACKUP.firstKey()));

ITEMS\_LOGS\_BACKUP.remove("" + ITEMS\_LOGS\_BACKUP.firstKey());

}

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

System.out.println("Log File ERROR.");

}

}

public static void add(String priority, EntryValue ev) {

BufferedWriter bw = null;

try {

bw = new BufferedWriter(new FileWriter(FILENAME, true));

bw.write(priority + "=" + ev);

bw.newLine();

bw.flush();

bw.close();

} catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

public static void retrieveLog() {

try {

BufferedReader br = new BufferedReader(new FileReader(FILENAME));

try {

getLines(br);

} catch (IOException e) {

// TODO Auto-generated catch block

System.out.println("File not available.");

}

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

System.out.println("File containing high scores not found.");

}

}

private static void getLines(BufferedReader br) throws IOException {

String line = br.readLine();

if (!(line == null)) {

A3Version1.ITEMS\_LOGS.put(line.substring(0, line.indexOf("=")), new EntryValue((line.substring(2))));

getLines(br);

}

}

public static void clearLog() {

PrintWriter out;

try {

out = new PrintWriter(FILENAME);

out.close();

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

package eecs2011a3;

class Node<k> {

String key;

int height;

Node<k> left, right;

public Node(String d) {

key = d;

height = 1;

}

}

package eecs2011a3;

public enum Parts {

BRAKES, OIL, FRONT\_LEFT\_TIRE, FRONT\_RIGHT\_TIRE, BACK\_LEFT\_TIRE, BACK\_RIGHT\_TIRE, TIRES, SHOCKS, SUSPSNSION, WASHER\_FLUID, LIGHTS

}

package eecs2011a3;

public interface Position<E> {

// returns the stored element

// throws exception if position is not valid

E getElement() throws IllegalStateException;

}

package eecs2011a3;

public class Readme {

}

package eecs2011a3;

public interface TBD {

void viewLog();

void saveLog();

void retreiveLog();

void addToLog();

void reorganize();

}

package eecs2011a3;

public class TEST\_CASES {

public static void runTestsVersion1() {

Log.clearLog(); // starts with a clean log (for testing)

A3Version1 a3 = new A3Version1();

EntryValue ev = a3.getInstances(Parts.FRONT\_RIGHT\_TIRE, Errors.P88\_HIGH\_PRESSURE);

EntryValue ev2 = a3.getInstances(Parts.BRAKES, Errors.B01\_NO\_BRAKES);

EntryValue ev3 = a3.getInstances(Parts.OIL, Errors.P99\_LOW\_PRESSURE);

EntryValue ev4 = a3.getInstances(Parts.BACK\_RIGHT\_TIRE, Errors.P88\_HIGH\_PRESSURE);

EntryValue ev5 = a3.getInstances(Parts.BACK\_RIGHT\_TIRE, Errors.P88\_HIGH\_PRESSURE);

System.out.println(ev);

System.out.println();

System.out.println(ev2);

System.out.println();

System.out.println(ev3);

System.out.println();

System.out.println(ev4);

System.out.println();

System.out.println(ev5);

a3.run();

Log.saveLog();

Log.saveLog();

}

public static void runTestsVersion2() {

}

}

package AutoInfoSys;

import java.util.List;

import java.util.ArrayList;

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

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

Position<E> parent = parent(p); // get Position of p's parent

if (parent == null) return null; // If parent of p is null the p must be root in which case it has no siblings

if (p == left(parent)) return right(parent); // If p is left child, return its right sibling

else return left(parent); // If p is right child, return its left sibling

}

public int numChildren(Position<E> p) {

int count = 0;

if (left(p) != null) count++; // if p has a left child, increment count

if (right(p) != null) count++; // if p has a right child, increment count

return count;

}

/\*\*

\* Accessor method returns iterable collection of p's children

\* @param p The node for which an iterable collection of its children is to be returned.

\* @return iterable collection of every <code>Position</code> of p's children.

\*/

public Iterable<Position<E>> children(Position<E> p) {

List<Position<E>> snapshot = new ArrayList<>(2); // an empty ArrayList serves as the snapshot

if (left(p) != null) snapshot.add(left(p)); // add children to the snapshot

if (right(p) != null) snapshot.add(right(p));

return snapshot;

}

}

package AutoInfoSys;

/\*\*

\* An abstract base class implementing the methods and providing

\* some functionality for the general tree interface.

\* This class will be used the make more concrete Tree structures.

\*/

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

/\*\*

\* <code>Position</code> p is an internal node if it

\* has more than zero (at least one) children.

\*/

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

/\*\*

\* <code>Position</code> p is an external node if it has zero children.

\*/

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

/\*\*

\* <code>Position</code> p is the root of the tree if it is equal to

\* return value of <code>root()</code> which is the root of the tree.

\*/

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

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

}

package AutoInfoSys;

import java.util.Iterator;

public class BalanceableBinaryTree<K,V> extends LinkedBinaryTree<Entry<K,V>> {

//----------------------- BST Node class -------------------------//

protected static class BSTNode<E> extends Node<E> {

int aux;

BSTNode(E e, Node<E> p, Node<E> lC, Node<E> rC) {

super(e, p, lC, rC);

aux = 0;

}

public int getAux() { return aux; }

public void setAux(int a) { aux = a; }

}//----------------------- End of BST Node class -------------------------//

// positional-based methods related to aux fields

public int getAux(Position<Entry<K,V>> p) { return ((BSTNode<Entry<K,V>>) p).getAux(); }

public void setAux(Position<Entry<K,V>> p, int v) { ((BSTNode<Entry<K,V>>) p).setAux(v); }

// Override factory method to produce BST Node and not a regular Node

protected Node<Entry<K,V>> createNode(Entry<K,V> e, Node<Entry<K,V>> parent, Node<Entry<K,V>> left, Node<Entry<K,V>> right) { return new BSTNode<>(e, parent, left, right); }

// Re-links parent node with oriented child node

private void relink(Node<Entry<K,V>> parent, Node<Entry<K,V>> child, boolean makeLeftChild) {

child.setParent(parent);

if (makeLeftChild) parent.setLeft(child);

else parent.setRight(child);

}

/\*\*

\* @Description: Rotates Position p (child) above its parent.

\* @param p Position (child) to be rotated above parent.

\*/

public void rotate(Position<Entry<K,V>> p) {

Node<Entry<K,V>> x = validate(p); // return p (child) as a validated Node

Node<Entry<K,V>> y = x.getParent(); // parent of p (child)

Node<Entry<K,V>> z = y.getParent(); // grandparent of p (child)

if (z == null) { // if p does'nt have grandparent then it becomes root/parent

root = x;

x.setParent(null);

} else relink(z, x, y == z.getLeft()); // make x direct child of y

// rotate x and y and transfer of middle subtree

if (x == y.getLeft()) {

relink(y, x.getRight(), true); // make x's right child y's left

relink(x, y, false); // make y x's right child

} else {

relink(y, x.getLeft(), false); // make x's left child y's right child

relink(x, y, true); // make y left child of x

}

}

// Performs tri-node restructuring of Position x with its parent/grandparent

public Position<Entry<K,V>> restructure(Position<Entry<K,V>> x) {

Position<Entry<K,V>> y = parent(x);

Position<Entry<K,V>> z = parent(y);

if ((x == right(y)) == (y == right(x))) { // matching alignments

rotate(y); // rotate y once

return y; // y is root of new subtree

} else {

rotate(x);

rotate(x); // rotate x twice

return x; // x is root of new subtree

}

}

@Override

public Iterator<Entry<K, V>> iterator() {

return null;

}

@Override

public Iterable<Position<Entry<K, V>>> positions() {

return null;

}

}

package AutoInfoSys;

/\*\*

\* An general Binary Tree interface, in which each node has at most two children.

\* This is a specialization of the Tree interface.

\*

\* This interface supports three additional accessor methods.

\*/

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

/\*\*

\* Accessor method that returns <code>Position</code> of p's left child or null.

\* @param p The internal node whose child is to be returned.

\* @return <code>Position</code> of p's left child or null if none exists.

\* @throws IllegalArgumentException if <code>Position</code> p is invalid.

\*/

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

/\*\*

\* Returns <code>Position</code> of p's right child or null.

\* @param p The internal node whose child is to be returned.

\* @return <code>Position</code> of p's left right or null if none exists.

\* @throws IllegalArgumentException if <code>Position</code> p is invalid.

\*/

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

/\*\*

\* Returns <code>Position</code> of p's sibling or null.

\* @param p The node whose sibling is to be returned.

\* @return <code>Position</code> of p's sibling or null if none exists.

\* @throws IllegalArgumentException if <code>Position</code> p is invalid.

\*/

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

}

package AutoInfoSys;

public interface Entry<K, V> {

K getKey();

V getValue();

}

package AutoInfoSys;

import java.util.Iterator;

/\*\*

\* Concrete implementation of a binary tree using a node-based, linked structure.

\*/

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

//-------------------- 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;

}

//---------------- GETTER ----------------------//

public E getElement() { return element; }

public Node<E> getParent() { return parent; }

public Node<E> getLeft() { return left; }

public Node<E> getRight() { return right; }

//----------------- SETTER -----------------------//

public void setElement(E e) { element = e; }

public void setParent(Node<E> node) { parent = node; }

public void setLeft(Node<E> node) { left = node; }

public void setRight(Node<E> node) { right = node; }

}//-------------------- End of Node ------------------//

/\*\*

\* Factory method to create new node with element e and return node instance

\* @param e element at this node.

\* @param p parent of this node.

\* @param l left child.

\* @param r right child.

\* @return new <code>Node</code> instance.

\*/

protected Node<E> createNode(E e, Node<E> p, Node<E> l, Node<E> r) {

return new Node<E>(e, p, l, r);

}

// LinkedBinaryTree attributes

protected Node<E> root; // root of tree

protected int size; // number of nodes in the tree

//-------------------- CONSTRUCTOR -------------------//

public LinkedBinaryTree() {

root = null; // initialize to null

size = 0; // initialize to zero

}

/\*\*

\* Protected utility method. Validates or ensures that given <code>Position</code> is a valid node.

\* @param p <code>Position</code> to be validated.

\* @return <code>Node</code> instance of validated <code>Position</code>.

\* @throws IllegalArgumentException if <code>Position</code> is invalid.

\*/

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

if (!(p instanceof Node)) throw new IllegalArgumentException("Not a valid position type");

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

if (node.getParent() == node) throw new IllegalArgumentException("p is no longer in the tree");

return node;

}

//---------------------------- accessor methods (not already implemented in AbstractBinaryTree)----------//

public int size() {

return size;

}

public Position<E> root() {

return root;

}

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

Node<E> node = validate(p);

return node.getParent();

}

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

Node<E> node = validate(p);

return node.getLeft();

}

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

Node<E> node = validate(p);

return node.getRight();

}

//-------------------- update methods supported by this class -----------------------//

/\*\*

\* Update method creates a root for an empty tree and stores e as the element.

\* <code>Position</code> of that root is returned.

\* @param e Element to be stored at new root.

\* @return <code>Position</code> of the new root.

\* @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;

}

/\*\*

\* Update method creates a new node, a new left child of Position p storing element e.

\* @param p The <code>Position</code> for which a new left child is created.

\* @param e Element to be stored at new <code>Position</code>.

\* @return <code>Position</code> of new node.

\* @throws IllegalArgumentException if the p 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;

}

/\*\*

\* Update method creates a new node, a new right child of Position p storing element e.

\* @param p The <code>Position</code> for which a new right child is created.

\* @param e Element to be stored at new <code>Position</code>.

\* @return <code>Position</code> of new node.

\* @throws IllegalArgumentException if the p 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;

}

/\*\*

\* Update method to replace element at p with new one.

\* @param p <code>Position</code> whose element is to be changed.

\* @param e new element.

\* @return previously stored element.

\* @throws IllegalArgumentException if p is invalid.

\*/

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

Node<E> node = validate(p);

E tmp = node.getElement();

node.setElement(e);

return tmp;

}

/\*\*

\* Update method that attaches trees t1 and t2 as left and right subtrees to leaf <code>Position</code> p,

\* and resets t1 and t2 to empty trees.

\* @param p leaf <code>Position</code> recieving subtrees t1 and t2.

\* @param t1 <code>LinkedBinaryTree</code> left subtree.

\* @param t2 <code>LinkedBinaryTree</code> right subtree.

\* @throws IllegalArgumentException if p is not a leaf.

\*/

public void attatch(Position<E> p, LinkedBinaryTree<E> t1, LinkedBinaryTree<E> t2) throws IllegalArgumentException {

Node<E> node = validate(p);

if (isInternal(p)) throw new IllegalArgumentException("p must be a leaf");

size += t1.size() + t2.size();

if (!t1.isEmpty()) { // attach t1 as left subtree of node

t1.root.setParent(node);

node.setLeft(t1.root);

t1.root = null;

t1.size = 0;

}

if (!t2.isEmpty()) { // attach t2 as right subtree of node

t2.root.setParent(node);

node.setRight(t2.root);

t2.root = null;

t2.size = 0;

}

}

/\*\*

\* Removes node at Position p and replaces it with its child, if any.

\* @param p <code>Position</code> that is being removed.

\* @return element that had been stored at <code>Position</code> 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 tmp = node.getElement();

node.setElement(null);

node.setLeft(null);

node.setRight(null);

node.setParent(node);

return tmp;

}

@Override

public Iterator<E> iterator() {

return null;

}

@Override

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

return null;

}

}

package AutoInfoSys;

import java.util.Iterator;

public class Main {

public static void main(String[] args) {

}

}

package AutoInfoSys;

/\*\*

\* An abstraction for node used in the Tree structure.

\* Every <code>Position</code> will represent a node and an element

\* will be stored at each <code>Position</code>.

\*/

public interface Position<E> {

/\*\*

\* Accessor method that returns the element at this <code>Position</code>.

\* @return Element of type <code>E</code> stored at this <code>Position</code>.

\* @throws IllegalStateException

\*/

E getElement() throws IllegalStateException;

}

package AutoInfoSys;

public class test {

public static void main(String[] args) {

System.out.print(Math.PI);

}

}

package AutoInfoSys;

import java.util.Iterator;

/\*\*

\* A general interface for a Tree in which nodes can have any number of children.

\* This Tree ADT uses the concept of <code>Position</code> to abstractly represent a node.

\* At each <code>Position</code> and element is stored.

\*/

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

/\*\*

\* Accessor method to return the Position of the root of the tree.

\* @return <code>Position</code> of the root of tree.

\*/

Position<E> root();

/\*\*

\* Accessor method to return the Position of the parent of child p.

\* @return <code>Position</code> of the parent p.

\* @throws IllegalArgumentException if <code>Position</code> p is invalid.

\*/

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

/\*\*

\* Accessor method to return an iterable collection of the children of Position p.

\* @return iterable collection of <code>Position</code>s of children of p.

\* @throws IllegalArgumentException if <code>Position</code> p is invalid.

\*/

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

/\*\*

\* Accessor method to return the number of children of Position p.

\* @return <code>int</code> representing the number of children of p.

\* @throws IllegalArgumentException if <code>Position</code> p is invalid

\*/

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

/\*\*

\* Query method to to determine whether p is an internal node.

\* @return <code>true</code> if p has at least one child.

\* @throws IllegalArgumentException if <code>Position</code> p is invalid

\*/

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

/\*\*

\* Query method to to determine whether p is an external node.

\* @return <code>true</code> if p has no children.

\* @throws IllegalArgumentException if <code>Position</code> p is invalid

\*/

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

/\*\*

\* Query method to to determine whether p is the root of the tree.

\* @return <code>true</code> if p is the root (has no parent).

\* @throws IllegalArgumentException if <code>Position</code> p is invalid

\*/

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

int size();

boolean isEmpty();

Iterator<E> iterator();

Iterable<Position<E>> positions();

}**TESTING LOG:**

01;0=Part: BRAKES,Error: B01\_NO\_BRAKES, Date: Tue Nov 21 14:07:16 EST 2017

88;3=Part: FRONT\_RIGHT\_TIRE,Error: P88\_HIGH\_PRESSURE, Date: Tue Nov 21 14:07:16 EST 2017

88;5=Part: BACK\_RIGHT\_TIRE,Error: P88\_HIGH\_PRESSURE, Date: Tue Nov 21 14:07:16 EST 2017

99;1=Part: OIL,Error: P99\_LOW\_PRESSURE, Date: Tue Nov 21 14:07:16 EST 2017

**README**

Clone in Eclipse © using <https://github.com/jskinner65/EECS2011a3> and run the main method.