# DD1339 Introduktion till datalogi 2013/2014

# Uppgift nummer: 5

# Namn: Marcus Larsson

# Grupp nummer: 5

# Övningsledare: Marcus Dicander

# Betyg: ..... Datum: .............. Rättad av: .......................................

# Exercise Binärt Träd

## BinTree

/\*\*

\* This is a Set that can only contain one of each item.

\* It is also a tree data structure. It is not yet a balanced tree.

\* @author Marcus

\*/

public class BinTree<T extends Comparable<T>>{

private int size;

private Node root;

private class Node<T extends Comparable<T>>{

public Node left,right;

public T data;

public Node(T data){

this.data=data;

this.left=null;

this.right=null;

}

}

/\*\*

\* Checks if the given element is in the tree.

\* @param element Element to search for.

\* @return True if tree contains the specified element. False if not.

\*/

public boolean contains(T element) {

if(element==null || size==0){

return false;

}

if(root.data.compareTo(element)==0){

return true;

}

Node next = root;

while(next!=null){

if(next.data.compareTo(element)==0){

return true;

} else if(next.data.compareTo(element)>0){

next=next.left;

} else if(next.data.compareTo(element)<0){

next=next.right;

}

}

return false;

}

/\*\*

\* Adds the given element to the tree. It will be added in it's correct place using the compareTo method from interface Comparable.

\* @param element The element to add into the tree.

\* @return True if the element was added and did not already existed in the tree. False if element already existed in tree.

\* @throws IllegalArgumentException if element is not of correct type.

\*/

public boolean add(T element) throws IllegalArgumentException{

if(element==null){

throw new IllegalArgumentException("Argument was null");

}

if(size==0){

root= new Node(element);

size++;

return true;

}

if(root.data.compareTo(element)==0){

return false;

}

//find where to put element

Node next = root;

while(next!=null){

if(next.data.compareTo(element)==0){

return false;

} else if(next.data.compareTo(element)>0){

if(next.left==null){

next.left= new Node(element);

size++;

return true;

}

next=next.left;

} else if(next.data.compareTo(element)<0){

if(next.right==null){

next.right= new Node(element);

size++;

return true;

}

next=next.right;

}

}

return false;

}

/\*\*

\*

\* @return The size of the tree. The number of elements in the tree.

\*/

public int size() {

return this.size;

}

/\*\*

\* The depth of the tree can be seen as the longest path from the root. The maximum number of steps to en up on a leaf.

\* @return The depth of the tree.

\*/

public int depth() {

return depth(this.root);

}

/\*\*

\* A private recursive help method for calculating the depth in the tree with the given root.

\* @param root The root of this tree

\* @return The depth of the tree with the given root.

\*/

private int depth(Node root){

if(root==null){

return 0;

}

return 1 + Math.max(depth(root.left), depth(root.right));

}

/\*\*

\* A leaf is an end node in the tree. All nodes that does not have any child is a leaf.

\* @return The number of leaves in the tree

\*/

public int leaves() {

return leaves(this.root);

}

/\*\*

\* A private recursive help method for calculating the number of leaves.

\* @param root The root of this tree

\* @return The number of leaves i the tree with the given root.

\*/

private int leaves(Node root){

if(root==null){

return 0;

}

if(root.left==null && root.right==null){

return 1;

}

return leaves(root.left)+leaves(root.right);

}

/\*\*

\* This will return a String representation of the tree in correct order. Smallest to largest.

\* Format will be: [element 1, element 2, .... , element n]

\* @return A sorted String representation of the tree.

\*

\*/

@Override

public String toString(){

StringBuilder result = new StringBuilder();

result.append("[");

result.append(inorder(this.root));

result.append("]");

return result.toString();

}

private String inorder(Node root){

if(root==null){

return "";

}

StringBuilder result = new StringBuilder();

result.append(inorder(root.left));

if(root.left!=null){

result.append(", ");

}

result.append(root.data.toString());

if(root.right!=null){

result.append(", ");

}

result.append(inorder(root.right));

return result.toString();

}

}

## BinTreeTest

/\*

\* This class is a test class for the data structure BinTree.

\*/

import java.util.Random;

import org.junit.Test;

import static org.junit.Assert.\*;

/\*\*

\*

\* @author Marcus

\*/

public class BinTreeTest {

/\*\*

\* Test to add different Strings that has same hasCode to make sure BinTree does not view them as same String.

\* Also test to check for null which should not be able to exist in the set.

\*/

@Test

public void testContains(){

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

assertFalse(tree.contains("FB"));

tree.add("FB");

assertTrue(tree.contains("FB"));

tree.add("Ea");

assertTrue(tree.contains("Ea"));

assertTrue(tree.contains("FB"));

assertFalse(tree.contains(null));

}

/\*\*

\* Test to add 10000 different Strings and checks so that all gets added.

\* Also tests to add empty string and two strings that has the same hashCode.

\*/

@Test

public void testAdd(){

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

assertTrue(tree.add("FB"));

assertFalse(tree.add("FB"));

assertTrue(tree.add("Ea"));

assertFalse(tree.add("Ea"));

assertTrue(tree.add(""));

assertFalse(tree.add(""));

for(int i=0;i<10000;i++){

assertTrue(tree.add("test"+i));

assertTrue(tree.contains("test"+i));

}

}

/\*\*

\* Test to add null to the tree which should throw IllegalArgumentException.

\*/

@Test (expected=IllegalArgumentException.class)

public void testAddNull(){

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

tree.add(null);

}

/\*\*

\* Test to add a random number of elements and checks if size is correct.

\*/

@Test

public void testSize() {

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

//test so that empty list has size 0.

assertEquals(0, tree.size());

// tree.removeFirst();

// assertEquals(0, tree.size());

//test a random number of elements is correct.

int numOfElements = (new Random().nextInt(100000))+1; //random between 1 and 100000

int count=0;

while(count<numOfElements){

tree.add("test"+count);

count++;

}

assertEquals(numOfElements, tree.size());

}

/\*\*

\* Test to add several elements and checks so that depth is what it is expected to be.

\* And not incremented when adding an element that should not add another depth level

\*/

@Test

public void testDepth(){

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

assertEquals(0 , tree.depth());

tree.add("5");

assertEquals(1, tree.depth());

tree.add("4");

assertEquals(2, tree.depth());

tree.add("6");

assertEquals(2, tree.depth());

tree.add("3");

assertEquals(3, tree.depth());

tree.add("2");

assertEquals(4, tree.depth());

}

/\*\*

\* Test to add several elements and checks so that number of leaves is correct.

\* And not incremented when adding a new element in the end of a previous leaf.

\*/

@Test

public void testLeaves(){

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

assertEquals(0, tree.leaves());

tree.add("5");

assertEquals(1, tree.leaves());

tree.add("3");

assertEquals(1, tree.leaves());

tree.add("6");

assertEquals(2, tree.leaves());

tree.add("2");

assertEquals(2, tree.leaves());

tree.add("4");

assertEquals(3, tree.leaves());

}

/\*\*

\* Test the toString method. Adds a few elements and compares toString to what it should be after a few elements added.

\* Test empty tree, 1 element, 2 element and 6 elements.

\*/

@Test

public void testToString(){

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

assertEquals("[]", tree.toString());

tree.add("5");

assertEquals("[5]", tree.toString());

tree.add("4");

assertEquals("[4, 5]", tree.toString());

tree.add("6");

tree.add("3");

tree.add("2");

tree.add("1");

assertEquals("[1, 2, 3, 4, 5, 6]", tree.toString());

}

}

# Exercise Tidskomplexitet

Contains:

Värstafallstiden är O(n). I värsta fall ligger alla element på en "rad" alltså inte är uppdelade. Då blir det som en sorterad länkad lista.

add:

Add måste kolla om elementet först finns. Men samtidigt kollar man var det i så fall skulle ligga. Samma som ovan så i värsta fall är det som en länkad lista. O(n).

Size:

Eftersom size ökas vid varje insättning och vi inte har någon metod för borttagning förlitar sig metoden på att instansvariabeln stämmer och har O(1).

Depth:

En rekursiv metod som har basfall då root är null. Alltså då det inte finns fler grenar. Den räknar längden av alla möjliga vägar och returnerar den längsta. För varje element returnerar metoden 1+maxDepth en gång. Tidskomplexiteten blir därför O(n).

Leaves:

På samma sätt som i depth måste denna besöka varje nod och räknar dem som inte har några barn. Tidskomplexitet blir då O(n).

toString:

Jag använder mig av StringBuilder för att optimera koden. Detta innebär därför också 1 besök hos varje nod och lägga till det i String builder blir O(n) som tidskomplexitet.

# Exercise Tidskomplexitet för Treap

Contains:

I en Treap är trädet balancerad och vi har garanterat O(log n) eftersom vi då får djupet till log n. Vi kommer inte få problemet att elementen kan hamna på "rad" och därmed få datastrukturen att bete sig som en länkad lista.

Add:

Eftersom trädet är balancerat hittar vi var det nya elementet ska sitta på O(log n). Genom en rekursiv metod kan vi samtidigt vid insättning då returnera den nya rooten i varje subträd efter rotering. Genom prioritetssiffror kan vi då samtidigt som insättning, i rekursionen, rotera trädet till att bli balancerat enligt prioriteten. Då har vi O(log n) för insättning också.