Aida Zhumabekova

a4/ BinaryTree

readme

Part I

**Called method:** System.out.println(tree.postorderResult());

**Input:** -pre treePre.data

**treePre.data content:**

“ +

2

null

null

3

null

null

”

**Output:**

2 3 + = 5

**Input:** -pre treePre2.data

**treePre2.data content:**

“ -

+

\*

2

null

null

+

3

null

null

4

null

null

\*

3

null

null

-

5

null

null

7

null

null

/

3

null

null

9

null

null

“

**Output:**

2 3 4 + \* 3 5 7 - \* + 3 9 / - = 8

**Input:** -pre treePre3.data

**treePre3.data content:**

“ \*

a

null

null

b

null

null

“

**Output:**

Invalid input. Can only perform evaluation of an arithmetic binary tree with numbers

a b \* = 0

**Input:** -post treePost.data

**treePost.data content:**

“ 2 4 1 - \* 2 7 + 3 / + ”

**Output:**

2 4 1 - \* 2 7 + 3 / + = 9

Part II

**Called methods:** tree = BinaryTree.readPostBinaryTree(new Scanner(new FileReader(args[1])));

System.out.println(tree);

**Input:** -post treePost.data

**Output:**

+

\*

2

null

null

-

4

null

null

1

null

null

/

+

2

null

null

7

null

null

3

null

null

**Input:** -post treePost2.data

**treePost2.data content:**

“ d e a - + a b + c \* \*”

**Output:**

\*

+

d

null

null

-

e

null

null

a

null

null

\*

+

a

null

null

b

null

null

c

null

null

**Input:** -post treePost3.data

**treePost3.data content:**

“ 5 6 ^ ”

**Output:**

Nothing happens because the program doesn't recognize the “^” sign. I intentionally set up my code so that I don’t build a tree from -post expression if it contains invalid operations. Consequently, I do not perform any methods if input contains invalid operations.

List of valid operations: “ +,\*,/,- ”

The reason I did that is because in instructions on noodle discussions you specified that we don’t have to worry about any other operations.

Part III

**Called method: tree.printTree(tree);**

**Method description:**

printTree() method accepts BinaryTree<String> tree and calls drawTree() method which accepts the root and the depth of the tree. drawTree() then recursively calls itself decreasing the initial depth of the tree, starting form the height of the tree, which is turned by getHeight(tree.root) method. It uses the ArrayList<Node<String>> to keep track of a queue of nodes to output. For every data of the node, which is null, program prints out “o”, else it prints out node.data. As soon as recursion reaches leaf nodes it stops after printing out every each of them.

**Input:** -post treePost.data

**Output:**

+

\* /

2 - + 3

o o 4 1 2 7 o o

**Input:** -post treePost2.data

**Output:**

\*

+ \*

d - + c

o o e a a b o o

**Input:** -pre treePre2.data

**Output:**

-

+ /

\* \* 3 9

2 + 3 - o o o o

o o 3 4 o o 5 7

Part IV

**Called methods:**

BinaryTree<String> combinedTree= new BinaryTree<String>(args[4], tree1, tree2);

combinedTree.printTree(combinedTree);

**Output:**

+

+ +

\* / 2 3

2 - + 3 o o o o

o o 4 1 2 7 o o

Method called: System.out.println(combinedTree.postorderResult());

**Output:**

2 4 1 - \* 2 7 + 3 / + 2 3 + + = 14

I have fully implemented each part of the assignment.