Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 4 REPORT

Azmi Utku SEZGİN
131044048

Course Assistant: b.koca@gtu.edu.tr

1 INTRODUCTION

1.1 Problem Definition

Part1:

Implementing general tree via binary tree and using post/level order traverse to search through the tree and add elements to nodes with specific value if they exist.

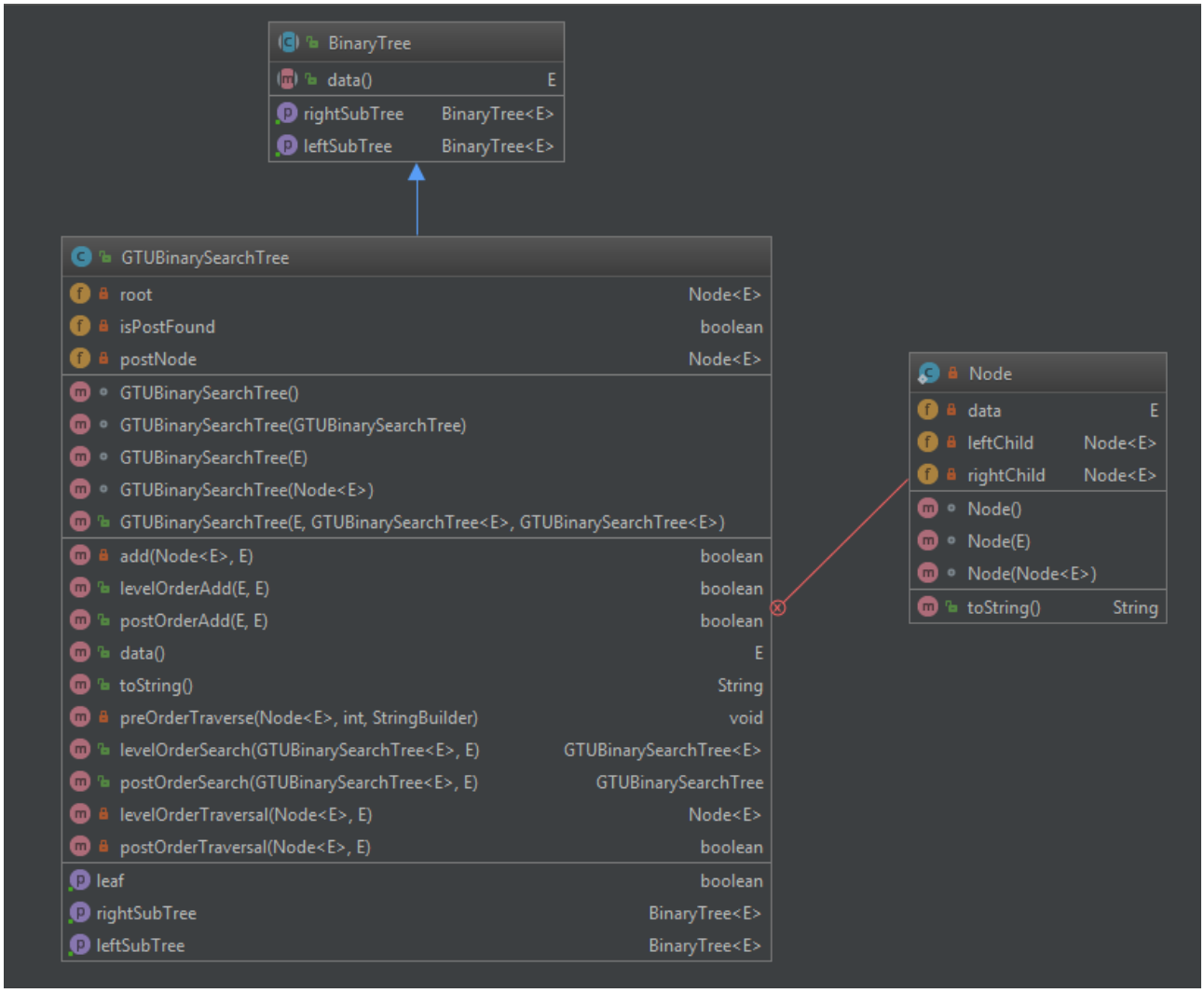
1.2 System Requirements

Java 8, Intellij Idea

To add elements to tree, you need to give an item that exist in the tree.

2 METHOD

2.1 Class Diagrams



2.2 Use Case Diagrams

Add use case diagrams if required.

2.3 Other Diagrams (optional)

Add other diagrams if required.

2.4 Problem Solution Approach

Part1:

GTUBinarySearchTree<E extends Comparable<E>> extends BinaryTree<E> class is a binary tree representation of general trees. (I realized that the name is binarysearchtree last second so didn't have time to change but it's a General Tree not binary search tree.) The parameter **E** must be **Comparable** to use this class as i used **compareTo()** method for checking items if they are equal or not. There are 2 public add methods and 1 helper add method in this class as following:

add(Node<E> localRoot, E childItem)
levelOrderAdd(E parentItem, E childItem)
postOrderAdd(E parentItem, E childItem)

What happens in these methods is, **levelOrderAdd** and **postOrderAdd** calls **levelOrderTraversal(Node<E> localRoot, E item)** and **postOrderTraversal(Node<E> localRoot, E item)** methods respectively and traverse through the tree to find parentItem, i will be talking about them more in detail after this section. If the item is found, they return the node they found the item, after we get the node, if it's null it means parentItem doesn't exist in the tree so the adding operation fails if the item does exist, then they call **add(Node<E> localRoot, E childItem)** method to insert elements to tree. If the left child of the node is empty, it adds the item to left node, otherwise it adds the item as right child note that the right node may not be empty, in that case it goes towards left untill the there's no right child.

levelOrderTraversal(Node<E> localRoot, E item) Method firstly pushes root into LinkedList, then untill there's no children left, it uses **pollLast()** method to get nodes from LinkedList, checks if it's equal to item, returns the node if the item is found otherwise looks for other child of the last popped up node but looks for only 1 left child while it gets all the right children as right children are representing the same level and left child means next level.
postOrderTraversal(Node<E> localRoot, E item) Method recursively gets to the last child, and then starts to go back through the tree to check if the item exists, there are 2 variables on our class **private boolean isPostFound** and **private Node<E> postNode** If the item is found, it assigns the node to **postNod**e and makes **isPostFound** true. And returns the isPostFound value at the end. This way **postOrderAdd** method can know that the item is found, get the node from **postNode** and call **add** method to finish the operation. After the operation done, **postOrderAdd** method makes **isPostFound false** and **postNode null**.

3 RESULT

3.1 Test Cases

First 5 screenshots are basicly creating a tree and adding elements to it in different order and the last **add()** calls are for testing if the parent item doesn't exist in the tree.

As for the 6th screenshot, there are 2 commented lines in main method and 1 commented line on both **levelOrderTraversal()** and **postOrderTraversal()** methods in tree class. The screenshot is the output of searching an item that i know it doesn't exist so that it could go through the whole tree and print elements out in that order.

3.2 Running Results

```
7 successfully added to tree, new overview:
10 5 15 20 30 40 41 12 13 7
15
100 successfully added to tree, new overview:
10 5 15 20 30 40 41 100 12 13 7
41
0 successfully added to tree, new overview:
10 5 15 20 30 40 41 0 100 12 13 7
Failed to add 42 to tree
10 5 15 20 30 40 41 0 100 12 13 7
```

```
10
5 successfully added to tree, new overview:
10 5
10
15 successfully added to tree, new overview:
10 5 15
15
20 successfully added to tree, new overview:
10 5 15 20
20
30 successfully added to tree, new overview:
```

```
6 successfully added to tree, new overview:
10 6
3 successfully added to tree, new overview:
10 6 3
5 successfully added to tree, new overview:
10 6 5 3
10 successfully added to tree, new overview:
10 6 5 10 3
20 successfully added to tree, new overview:
```

```
12 successfully added to tree, new overview:
10 6 5 15 10 3 20 12
35 successfully added to tree, new overview:
10 6 5 15 35 10 3 20 12
88 successfully added to tree, new overview:
10 6 5 15 35 88 10 3 20 12
21 successfully added to tree, new overview:
10 6 5 15 35 88 10 3 20 12 21
8 successfully added to tree, new overview:
10 6 5 15 35 88 10 3 20 12 21 8
Failed to add 42 to tree
10 5 15 20 30 40 41 0 100 12 13 7
```

```
Treel Pre-order from main:
10 5 15 20 30 40 41 0 100 12 13 7
Tree2 Pre-order from main:
10 6 5 15 35 88 10 3 20 12 21 8
```

```
Treel Pre-order:
10 5 15 20 30 40 41 0 100 12 13 7
Post Order Test
40
0
41
30
100
20
7
13
12
15
5
10
Tree2 Pre-order:
10 6 5 15 35 88 10 3 20 12 21 8
Level Order Test
10
6
3
8
5
10
20
15
12
35

Process finished with exit code 0
```