GIT Department of Computer Engineering
CSE 222/505 - Spring 2022
Homework 7 Report

Emre YILMAZ
1901042606

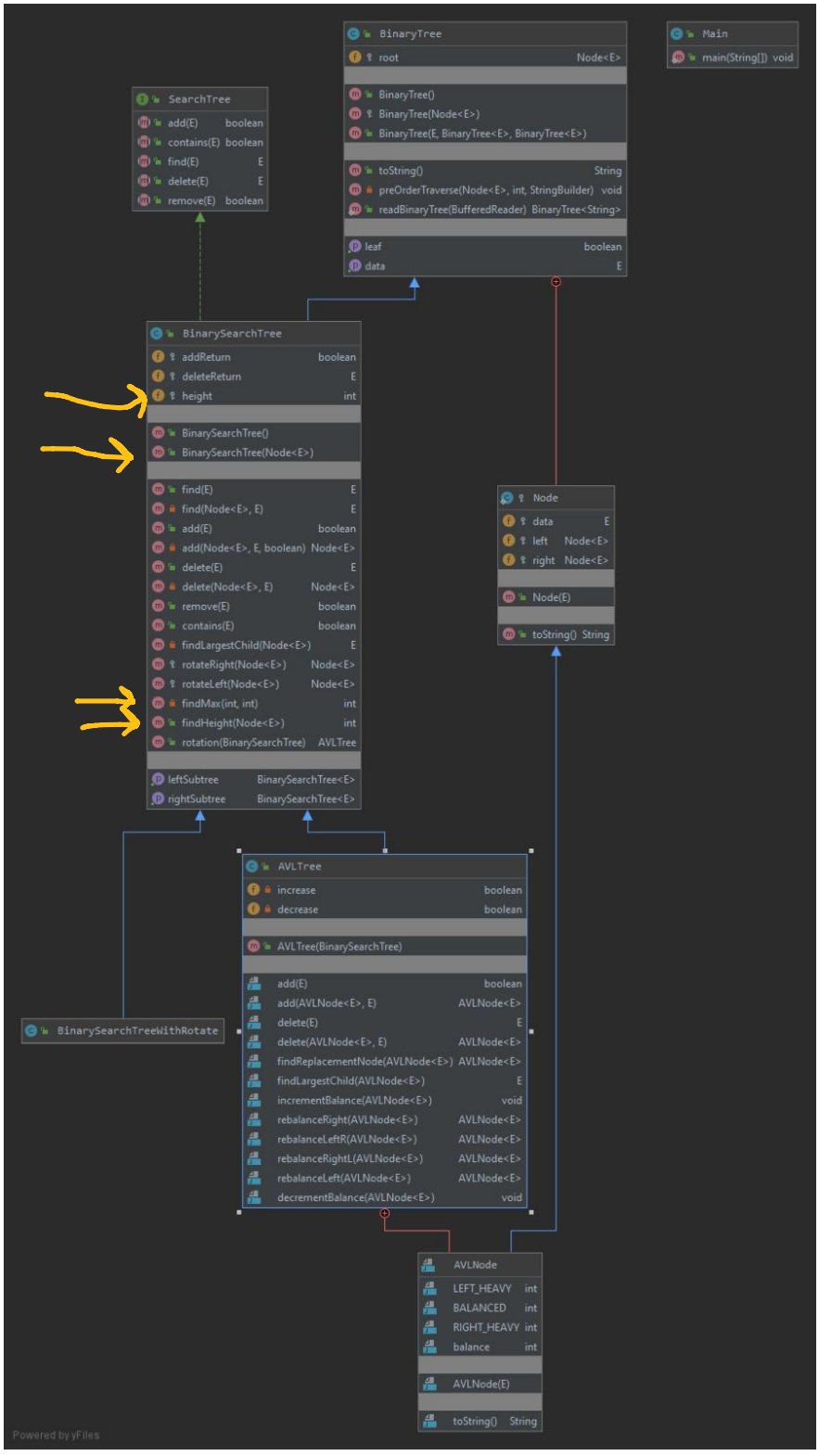
1. SYSTEM REQUIREMENTS

In the **first** question, we are asked to write a method that takes a binary tree and an array of items as input, and it returns a binary search tree (BST) as output. The array contains n unique items which are mutually comparable. The method should build a binary search tree of n nodes. The binary search tree should contain the items. The structure of the binary search tree should be same as the structure of the binary tree. After that, we have to analysis the algorithm.

In the **second** question, we are asked to write a method that takes a binary search tree (BST) as a parameter and returns the AVL tree obtained by rearranging the BST. The method should convert the BST into an AVL tree by using rotation operations. After that, we must analysis the algorithm.

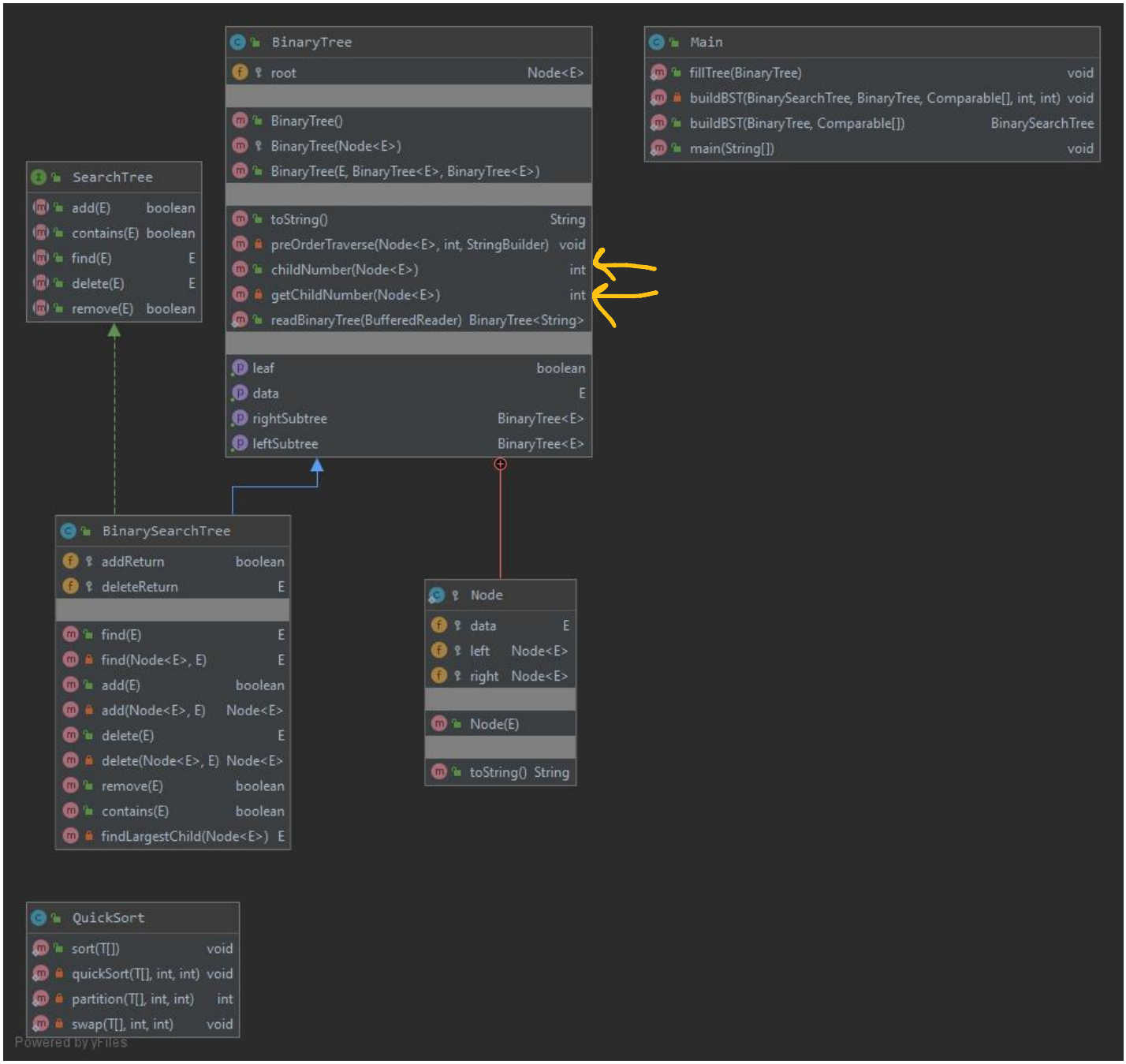
2. **USE CASE AND CLASS DIAGRAMS**

Q-2)



<u>Yellow methods and fields are NEW. Inserted by me</u>

Q-1)



<u>Yellow methods and fields are NEW. Inserted by me</u>

3. PROBLEM SOLUTION APPROACH

In the **first** question, the array is sorted with Quick Sort algorithm firstly. So, it is easy to build an algorithm now. Then, a recursive helper method is called with an index value of the root. This function will traverse the Binary Tree. If it goes towards the left, it counts the left children of the *localRoot*. It decreases the index value by the number of left children. Thus, it correctly finds the next array element to be added to BST. Then it calls 2 recursive function that one of them will go towards left, other will go towards right with new *localRoot* that is *leftSubTree* of the current *localroot*. The base case is *localRoot*=null that means the tree is over. The case of going to the right is very similar going left case. This time, it increases the index value by the number of right children. Thus, it correctly finds the next array element to be added to BST. To find children number, a method *childNumber* is added to Binary Tree class

In the **second** question, a method is added to BST Class. This method gets a BST input, balances it and returns as AVL tree.

First, we find the right and left heights of the root and find out which side the balance is broken. Then, which side is unbalanced, we go down one node and see which side is unbalanced again. Then we perform rotation operations according to cases such as left-left, right-left. This process continues until the root balance is found iterately.

Some important changes have been made to this method. Added *height* variable. The *add* method has been modified to increase the *height* when adding to a node that has no children. This was handled by adding the boolean *height* parameter to the helper recursive *add* function. *GetLeftSubtree* and *GetRightSubtree* methods moved to Binary Search Tree class from the Binary Tree class. *Constructor* with root parameter of type *Node<E>* has been added to Binary Search Tree class to use *GetLeftSubtree* and *GetRightSubtree* methods

4. TEST CASES

Q-1.)

Different randomly written 6 binary trees are tested. If you want, you can create a new tree and new array to test. Note: Arrays created in order. But it does not matter since there is QuickSort algorithm in the BuildBST method.

Q-2.)

There is a loop that generates random numbers and adds to the binary search tree. Then, this tree calls the *rotation* method to balance. You can run the program as many times as you want. It will build BST with different random numbers each time and then balance it.

5. RUNNING AND RESULTS

Q-1.)

1.

```
BINARY TREE:

3
  3
    3
      3
        null
        null
      null
    null
  3
    3
      3
        3
          null
          null
        null
      null
    null
```

```
BST:

4
  3
    2
      1
        null
        null
      null
    null
  8
    7
      6
        5
          null
          null
        null
      null
    null
```

2.

```
BINARY TREE:

3
  3
    null
    null
  3
    null
    3
      null
      3
        3
          null
          null
        null
```

```
BST:

2
  1
    null
    null
  3
    null
    4
      null
      6
        5
          null
          null
        null
```

3.

```
BINARY TREE:

3
  3
    3
      3
        null
        null
      null
    null
  3
    null
    null
```

```
BST:

4
  3
    2
      1
        null
        null
      null
    null
  5
    null
    null
```

4.

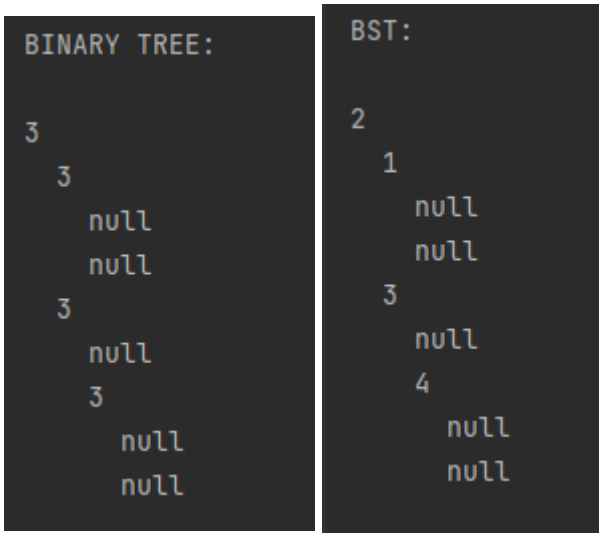
```
BINARY TREE:

3
  3
    3
      null
      null
    3
      3
        null
        3
          null
          null
      null
  3
    null
    3
      3
        null
        null
      3
        3
          null
          null
        null
```

```
BST:

6
  2
    1
      null
      null
    5
      3
        null
        4
          null
          null
      null
  7
    null
    9
      8
        null
        null
      11
        10
          null
          null
        null
```

5.


BINARY TREE:
3
3
null
null
3
null
3
null
null
BST:
2
1
null
null
3
null
4
null
null

6.


BINARY TREE:
3
3
3
null
3
null
null
null
null
BST:
4
3
1
null
2
null
null
null
null

Q-2.)

1.

```
Unmodified:

0
  null
  26
    null
    84
      null
      173
        null
        180
          175
            null
            null
          null
```

```
Modified:

84
  26
    0
      null
      null
    null
  173
    null
    180
      175
        null
        null
      null
```

2.

```
Unmodified:

-121
  null
  98
    -43
      null
      82
        null
        83
          null
          null
    null
```

```
Modified:

82
  -43
    -121
      null
      null
    null
  98
    83
      null
      null
    null
```

3.

```
Unmodified:

71
  14
    null
    22
      null
      50
        27
          null
          39
            34
              33
                30
                  28
                    null
                    null
                  null
                null
              null
            null
        null
  null
```

```
Modified:

34
  27
    22
      14
        null
        null
      null
    33
      30
        28
          null
          null
        null
      null
  39
    null
    50
      null
      71
        null
        null
```

4

```
Unmodified:

23
  1
    null
    5
      null
      6
        null
        19
          13
            11
              9
                null
                null
              null
            null
          null
  null
```

```
Modified:

13
  6
    5
      1
        null
        null
      null
    11
      9
        null
        null
      null
  19
    null
    23
      null
      null
```

5.

```
Unmodified:

99
  93
    44
      null
      84
        null
        90
          87
            85
              null
              null
            null
          null
    null
  null
```

```
Modified:

90
  84
    44
      null
      null
    87
      85
        null
        null
      null
  93
    null
    99
      null
      null
```

Analysis:

Q.1

- **childNumber** method:
  Best Case: Θ(logn)
  Worst Case: Θ(n)

- Main recursive **buildBST** method:
  Best Case:
  - T(n) = 2T(n/2) + logn
  - T(1) = Θ(1)
  - T(n) = Θ($\log^2 n$)

  Worst Case:

  - T(n) = 2T(n/2) + n
  - T(1) = Θ(1)
  - T(n) = Θ(nlogn)

Q.2

- **getRightSubTree** and **getLeftSubTree** method (This method calls the method below) :
  Best Case: Θ(logn) -- (The case that the BST is balanced)
  Worst Case: Θ(n) -- (The case that the BST is Θ(n))

- **getHeight** method
  Best Case: Θ(logn) -- (The case that the BST is balanced)
  Worst Case: Θ(n)) -- (The case that the BST is Θ(n),

- Main **rotation** method:
  Best Case: Θ(logn) -- (The case that the BST is balanced, there is no need to rotate)
  Worst Case: Θ($\log^2 n$) -- (The case that the BST is Θ(n), it needs logn rotation)