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

Hasan Mutlu
1801042673

SYSTEM REQUIREMENTS

Functional Requirements:

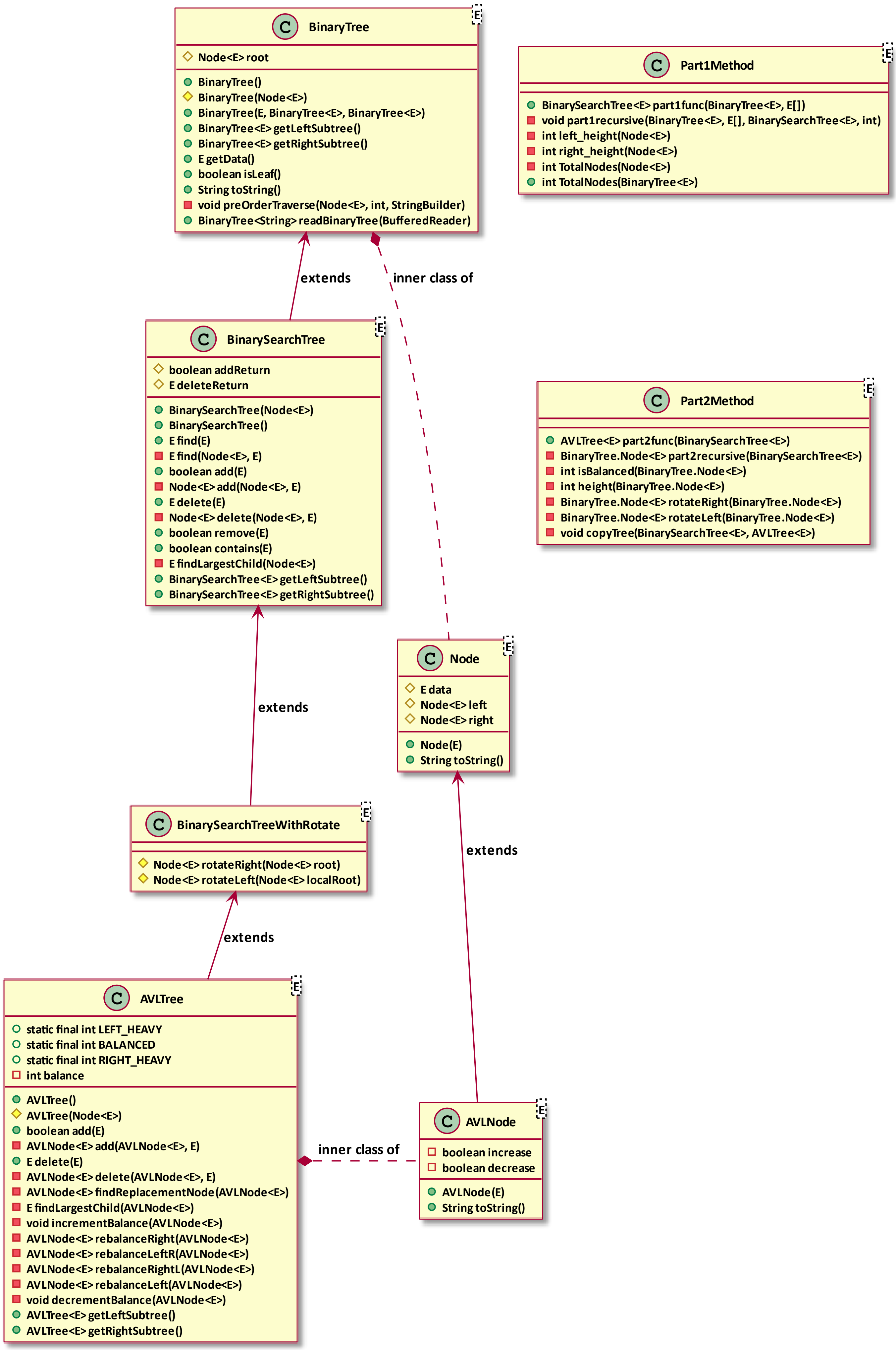
- ➔ User should be able to see the result of hard-coded test cases.

Non-Functional Requirements:

- ➔ Implementation: The programs shall be implemented using VSCode, Ubuntu 18.04 WSL and Java 11.
- ➔ Compiling and Running: The programs should be compiled and run with following commands:
- ➔ -$ javac *.java PartX.java
- ➔ -$ java PartX
- ➔ Efficiency: The algorithms must run as efficiently as possible.
- ➔ Relaibility: The algorithms and the programs must run reliably, should handle every possible valid usage.

CLASS DIAGRAMS

- ➔ Part1 uses BinaryTree and BinarySearchTree on the left.
- ➔ Part2 uses all classes on the left.
- ➔ The data structures on the left are implemented from book and several overriden methods are added as well.

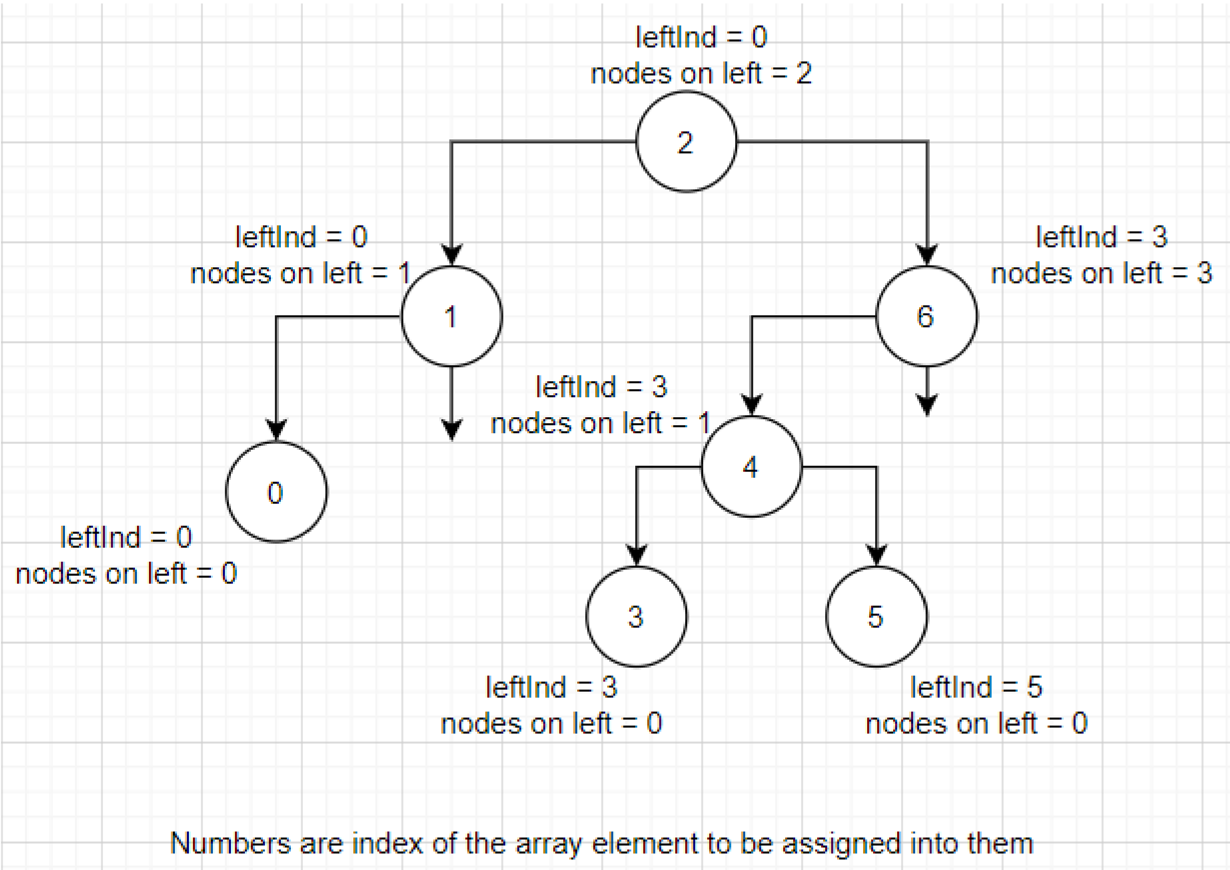

BinaryTree
Node<E> root
BinaryTree()
BinaryTree(Node<E>)
BinaryTree(E, BinaryTree<E>, BinaryTree<E>)
BinaryTree<E> getLeftSubtree()
BinaryTree<E> getRightSubtree()
E getData()
boolean isLeaf()
String toString()
void preOrderTraverse(Node<E>, int, StringBuilder)
BinaryTree<String> readBinaryTree(BufferedReader)
Part1Method
BinarySearchTree<E> part1func(BinaryTree<E>, E[])
void part1recursive(BinaryTree<E>, E[], BinarySearchTree<E>, int)
int left_height(Node<E>)
int right_height(Node<E>)
int TotalNodes(Node<E>)
int TotalNodes(BinaryTree<E>)
extends
inner class of
BinarySearchTree
boolean addReturn
E deleteReturn
BinarySearchTree(Node<E>)
BinarySearchTree()
E find(E)
E find(Node<E>, E)
boolean add(E)
Node<E> add(Node<E>, E)
E delete(E)
Node<E> delete(Node<E>, E)
boolean remove(E)
boolean contains(E)
E findLargestChild(Node<E>)
BinarySearchTree<E> getLeftSubtree()
BinarySearchTree<E> getRightSubtree()
Part2Method
AVLTree<E> part2func(BinarySearchTree<E>)
BinaryTree.Node<E> part2recursive(BinarySearchTree<E>)
int isBalanced(BinaryTree.Node<E>)
int height(BinaryTree.Node<E>)
BinaryTree.Node<E> rotateRight(BinaryTree.Node<E>)
BinaryTree.Node<E> rotateLeft(BinaryTree.Node<E>)
void copyTree(BinarySearchTree<E>, AVLTree<E>)
extends
Node
E data
Node<E> left
Node<E> right
Node(E)
String toString()
BinarySearchTreeWithRotate
Node<E> rotateRight(Node<E> root)
Node<E> rotateLeft(Node<E> localRoot)
extends
extends
AVLTree
static final int LEFT_HEAVY
static final int BALANCED
static final int RIGHT_HEAVY
int balance
AVLTree()
AVLTree(Node<E>)
boolean add(E)
AVLNode<E> add(AVLNode<E>, E)
E delete(E)
AVLNode<E> delete(AVLNode<E>, E)
AVLNode<E> findReplacementNode(AVLNode<E>)
E findLargestChild(AVLNode<E>)
void incrementBalance(AVLNode<E>)
AVLNode<E> rebalanceRight(AVLNode<E>)
AVLNode<E> rebalanceLeftR(AVLNode<E>)
AVLNode<E> rebalanceRightL(AVLNode<E>)
AVLNode<E> rebalanceLeft(AVLNode<E>)
void decrementBalance(AVLNode<E>)
AVLTree<E> getLeftSubtree()
AVLTree<E> getRightSubtree()
inner class of
AVLNode
boolean increase
boolean decrease
AVLNode(E)
String toString()

PROBLEM SOLUTION APPROACH

In part 1, I firstly implemented the BinaryTree and BST from the book. I thought of a solution but there were some external issues, mostly about class casting etc. So I added couple of overriden methods in BinarySearchTree to get rid of those issues. I created a class for the main method and its helpers. In main method, a BST is created, the array is sorted because it is supposed to be sorted for my solution, and a helper recursive method is called. There are also 4 other methods to help recursive method.

The recursive method gets one BinaryTree, on BST, the array and a start index information. It counts the number of nodes in the left side of the BinaryTree. It adds this number with the starting index and assgins the element in that resulting index from array to the top of the current BST tree/subtree. After that, it calls the recursive method for left and right subtrees. The starting index stays the same for left subtree's call, but it is incremented by number of nodes on the left + 1 for right subtree's call. By this way, array is virtually divided in every recursive call.

Explanation Figure:



In part 2, I implemented the data structures from the book, again. I fixed the same issues as part1. I created a class for the main method and its helpers. The main method first calls a recursive method to rearrange the BST, creates an empty AVLTree and copies the BST into AVLTree.

The recursive method gets a BST tree/subtree. It first checks if the tree is balanced, by counting height of nodes in either side of the tree. If this balance value is not between -1 and 1, the tree is heavy on one side. It rotates the tree in opposite direction to rebalance. It repeats this process until it gets balanced. After that, it calls the same recursive calls for its subtrees. So the tree gets balanced from top to bottom.

COMPLEXITY ANALYSIS

Part1 Method:

- ➔ Firstly, since the array is not sorted, it sorts the array using Java’s default Arrays.sort() method. It is O(n logn).
- ➔ Then the recursive method is called:
- ➔ Recursive method covers the entire BinaryTree elements. So it is called n times.
- ➔ Inside, it calls countNodes method. This method covers entire elements of either subtrees. So it runs (n-1) times.
- ➔ Combining these two clauses, recursive method runs in Teta(n^2) time.
- ➔ Returning to main method, recursive method has the biggeest running time, so it is the determinative factor. The method runs in Teta(n^2) time in overall.

Part2 Method:

- ➔ Firstly, copyTree method covers entire BST, so it runs n times.
- ➔ Before that, recursive method is called:
- ➔ The recursive method covers the entire BST, so it is called at least n times.
- ➔ Inside, there is a while loop. This loop continues until either side of the tree gets balanced. In best case, the tree is already balanced so it runs once. While balancing, the weight of one side is transferred to the other side. So in worst case, it runs (n/2) times.
- ➔ Inside the while loop, recursive isBalanced method is called. This method covers entire subtrees, so it runs (n-1) times.
- ➔ As a result of these three clauses, in best case, the while loop is only ran once. Main recursive method and isBalanced methods run in Teta(n) time. Combination of them makes Teta(n^2) time.
- ➔ In worst case, while loop runs for (n/2) times, which makes Teta(n). Combining all three, it runs in Teta(n^3) time.
- ➔ Overall, it runs in Teta(n^3) time.

TEST CASES

Test Case #	Test Case Description	Test Data	Expected Result	Actual Result	Pass/Fail
Part1 Test1	Random unbalanced binary tree	An array of 7 randomly generated integers	BinarySearchTree with array elements in the form of BinaryTree	As Expected	Pass
Part1 Test2	Linearly unbalanced binary tree	An array of 7 randomly generated integers	BinarySearchTree with array elements in the form of BinaryTree	As Expected	Pass
Part1 Test3	Linearly unbalanced binary tree	An array of 7 randomly generated integers	BinarySearchTree with array elements in the form of BinaryTree	As Expected	Pass
Part2 Test1	Linearly inbalanced BinarySearchTree	Elements: 10, 20, 30, 40, 50, 60, 70, 80	A well balanced AVLTree with the elements of BinarySearchTree	As Expected	Pass
Part2 Test2	Linearly inbalanced BinarySearchTree	Elements: 10, 20, 30, 40, 50, 60, 70, 81	A well balanced AVLTree with the elements of BinarySearchTree	As Expected	Pass
Part2 Test3	Already Balanced Binary Search Tree	Elements: 10, 20, 30, 50, 70, 80, 90	AVLTree exactly same as the BST	As Expected	Pass

RUNNING AND RESULTS

Part1:

```
TEST 1: Random unbalanced binary tree
The binary tree:
0
  0
    0
      null
      null
    null
  0
    0
      0
        null
        null
      0
        null
        null
    null

The random numbers in the array:
2 4 50 60 72 81 96
```

```
The binary search tree:
50
  4
    2
      null
      null
    null
  96
    72
      60
        null
        null
      81
        null
        null
    null
```

```
TEST 2: Linearly unbalanced binary tree
The binary tree:
0
  null
  0
    null
    0
      null
      0
        null
        0
          null
          0
            null
            0
              null
              null

The random numbers in the array:
3 25 52 54 79 87 97
```

```
The binary search tree:
3
  null
  25
    null
    52
      null
      54
        null
        79
          null
          87
            null
            97
              null
              null
```

```
TEST 3: Balanced binary tree
The binary tree:
0
  0
    0
      null
      null
    0
      null
      null
  0
    0
      null
      null
    0
      null
      null


The random numbers in the array:
11 15 24 50 58 70 95
```

```
The binary search tree:
50
  15
    11
      null
      null
    24
      null
      null
  70
    58
      null
      null
    95
      null
      null
```

Part2:

```
TEST1: Linearly inbalanced BinarySearchTree
10
  null
  20
    null
    30
      null
      40
        null
        50
          null
          60
            null
            70
              null
              80
                null
                null
```

```
The AVL Tree:
40
  20
    10
      null
      null
    30
      null
      null
  60
    50
      null
      null
    70
      null
      80
        null
        null
```

```
TEST2: Inbalanced Binary Search Tree
50
  10
    null
    40
      30
        20
          null
          null
        null
      null
  60
    null
    70
      null
      80
        null
        null
```

```
AVL Tree
40
  20
    10
      null
      null
    30
      null
      null
  70
    60
      50
        null
        null
      null
    80
      null
      null
```

```
TEST3: Already Balanced Binary Search Tree
50
  20
    10
      null
      null
    30
      null
      null
  80
    70
      null
      null
    90
      null
      null
```

```
The AVL Tree
50
  20
    10
      null
      null
    30
      null
      null
  80
    70
      null
      null
    90
      null
      null
```