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

Yusuf Arslan
200104004112

1. System Requirement

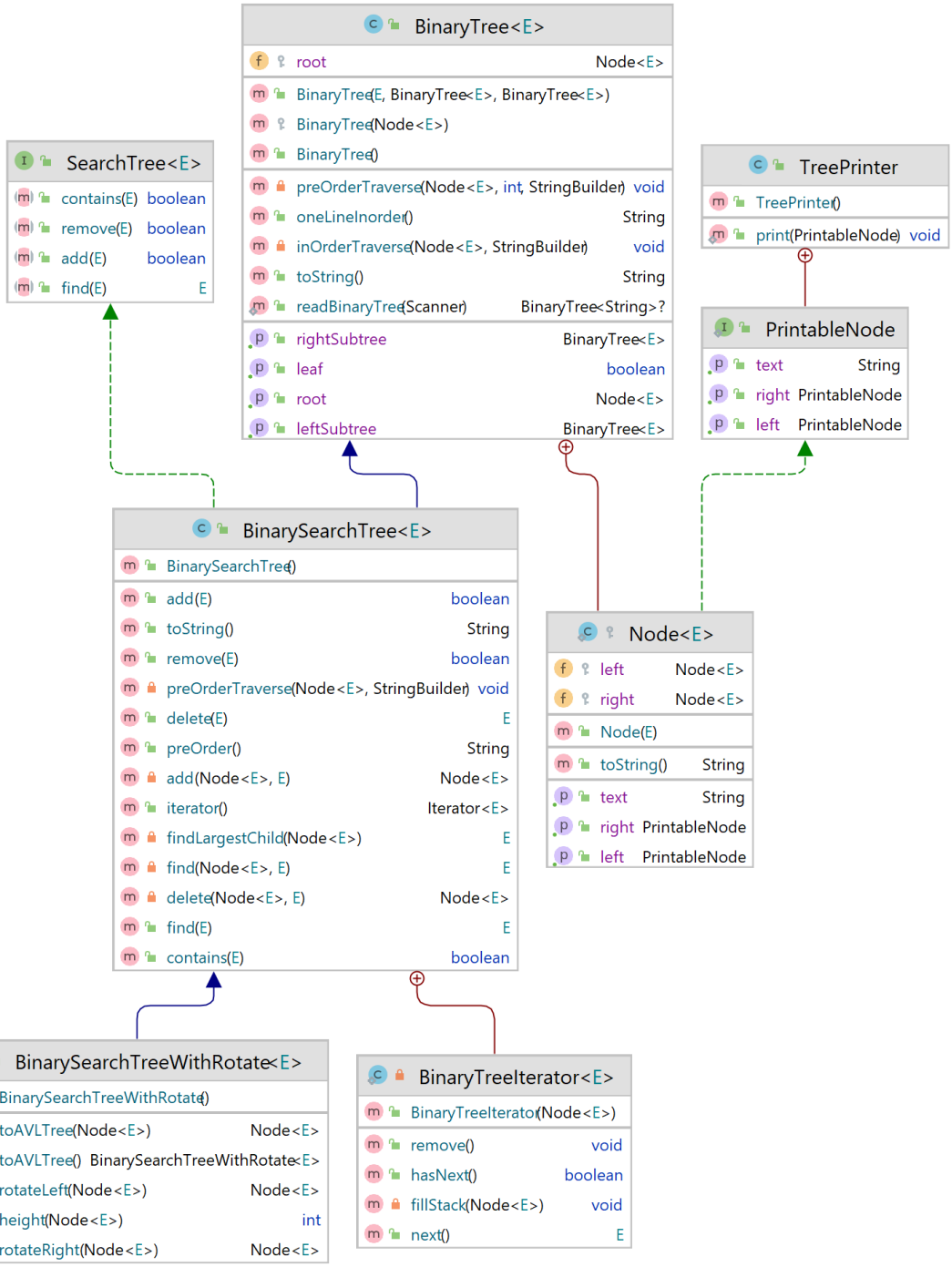
First of all the operating system should have Java virtual machine (JVM), Java development kit (JDK) and ‘at least’ Java runtime environment (JRE) – 11 in a linux distribution.

This application not interactive. User should compile and run program to see results of testing `toBinarySearchTreeWithArrayElements()` and `toAVLTree()` methods.

But, user is free to change tests which are in the Driver class.

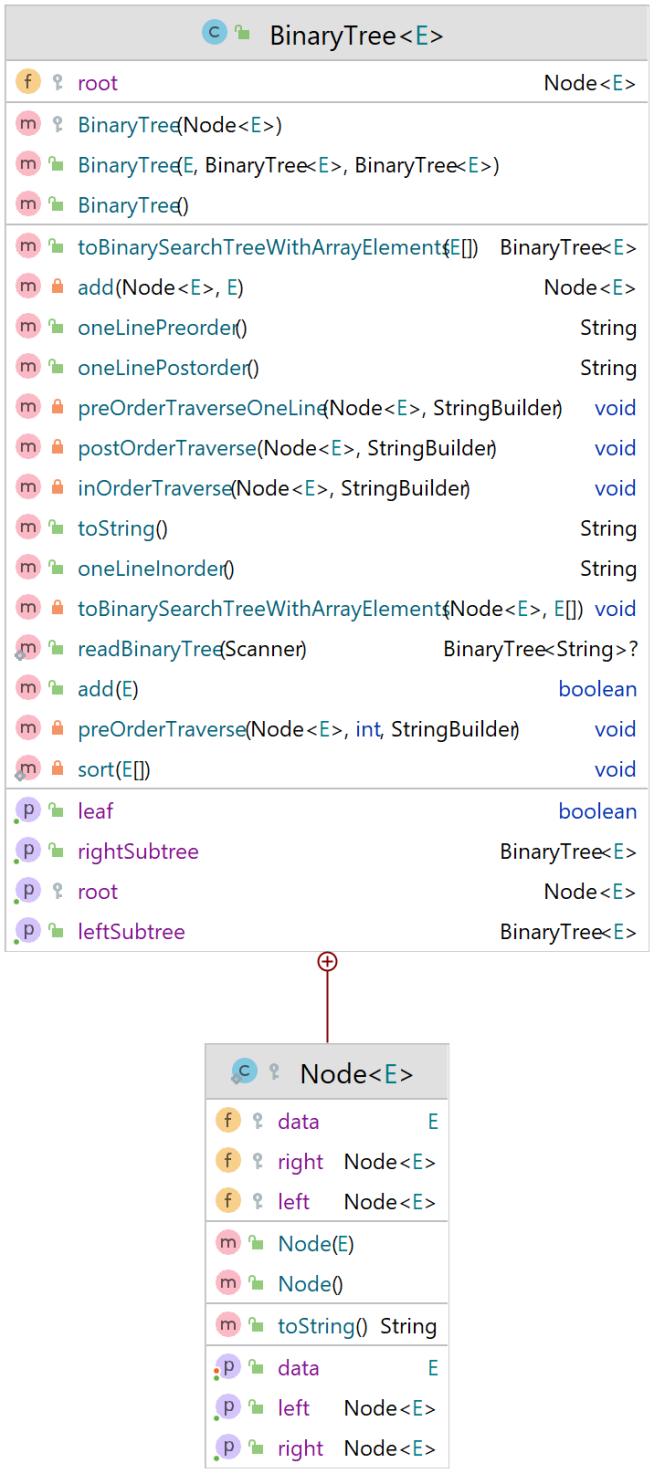
2. Class Diagrams

1. `toBinarySearchTreeWithArrayElements()`



(Implementations are in **`BinarySearchTreeWithRotate`** class)

2.toAVLTree()


BinaryTree<E>
root Node<E>
BinaryTree(Node<E>)
BinaryTree(E, BinaryTree<E>, BinaryTree<E>)
BinaryTree()
toBinarySearchTreeWithArrayElements(E[]) BinaryTree<E>
add(Node<E>, E) Node<E>
oneLinePreorder() String
oneLinePostorder() String
preOrderTraverseOneLine(Node<E>, StringBuilder) void
postOrderTraverse(Node<E>, StringBuilder) void
inOrderTraverse(Node<E>, StringBuilder) void
toString() String
oneLineInorder() String
toBinarySearchTreeWithArrayElements(Node<E>, E[]) void
readBinaryTree(Scanner) BinaryTree<String>?
add(E) boolean
preOrderTraverse(Node<E>, int, StringBuilder) void
sort(E[]) void
leaf boolean
rightSubtree BinaryTree<E>
root Node<E>
leftSubtree BinaryTree<E>
Node<E>
data E
right Node<E>
left Node<E>
Node(E)
Node()
toString() String
data E
left Node<E>
right Node<E>

3. Functions in Details and Run-time Complexities

1. `toBinarySearchTreeWithArrayElements()`

This method takes an array of comparable elements,
sorts that array, initialize INDEX with 0 and invokes **`private toBinarySearchTreeWithArrayElements()`** method to set elements in array to the binary tree in order to protect struct of binary tree and convert it to a binary search tree.

```
public BinaryTree<E> toBinarySearchTreeWithArrayElements(E[] array)
{
    // Sort the array first.
    sort(array);

    // Initialize static INDEX variable to 0.
    INDEX = 0;
    // Travers in order and set the element in the array to the tree.
    toBinarySearchTreeWithArrayElements(root, array);
    return this;
}
```

Let's look at the helper methods.

`Private sort()`

This method sort the array with selection sort algorithm.

```
private static <E extends Comparable<E>> void sort(E[] array)
{
    for (int i = 0; i < array.length; i++) {
        for (int j = i + 1; j < array.length; j++) {
            if (array[i].compareTo(array[j]) > 0) {
                E temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
    }
}
```

`private toBinarySearchTreeWithArrayElements()`
This method, takes the root of binary tree as a parameter and the array of elements. It traverses the binary tree in order and sets the element to the tree.

```
private void toBinarySearchTreeWithArrayElements(Node<E> localNode, E[] array) {
    // A Binary search tree suppose to be sorted in in-order travarsal.
    // Therefor if we in order traverse the tree and set the elements of sorted array to the tree,
    // the tree will be a binary search tree with same structure that invoked by this method.
    if (localNode != null)
    {
        toBinarySearchTreeWithArrayElements(localNode.left, array);
        // Set the element in the array to the tree and increase INDEX by one.
        localNode.data = array[INDEX++];
        toBinarySearchTreeWithArrayElements(localNode.right, array);
    }
}
```

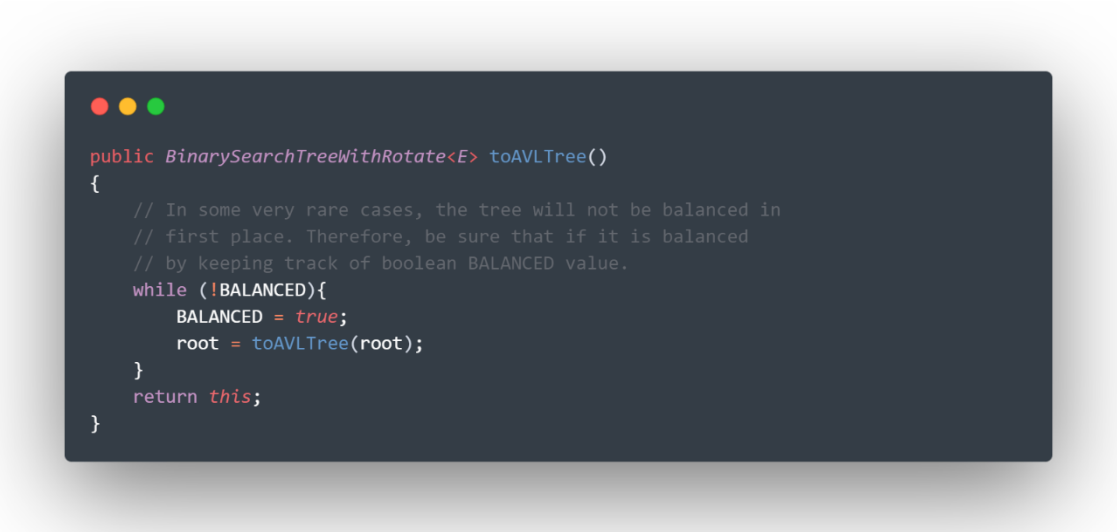
Run-time complexity:

To perform sorting, we need 1 + 2 + … + n – 1 comparision, which is $\Theta(n^2)$.

`private toBinarySearchTreeWithArrayElements()`, traversing the tree in order and sets the elements of the array to the tree, and it is $\Theta(n)$. Therefore, the time complexity of the ***`toBinarySearchTreeWithArrayElements`*** method is $\Theta(n^2)$.

2. `toAVLTree()`

This method, converts the binary search tree to an AVL tree by using left or right rotation operations.



```
public BinarySearchTreeWithRotate<E> toAVLTree()
{
    // In some very rare cases, the tree will not be balanced in
    // first place. Therefore, be sure that if it is balanced
    // by keeping track of boolean BALANCED value.
    while (!BALANCED){
        BALANCED = true;
        root = toAVLTree(root);
    }
    return this;
}
```

Unlike *AVL Tree data structures*, binary search tree can be in any shape and this shape can be disproportionate. (One branch of the tree may be much longer). So, the doing left and right rotation operations once, can not be enough in some cases. Therefore, we will call the **`private toAVLTree()`** method inside the while loop and be sure the tree is balanced by iterating from leaf to root over and over again. But, although the **`private toAVLTree()`** method is in the while loop, it will be invoked a few times (2, 3, 4) in most cases. Therefore, it will not affect time complexity on average too much.

`private toAVLTree()`

In this method, I traversed the tree post order. Then in each node, I calculated the left and right height of subtrees, if one of the heights is greater than 2 or more than the other height, to obtain if it is the left-left, left-right, right-right, or right-left tree, I calculated height of childs as well. If it is left-right or right-left, we need two rotations, otherwise, one rotation will be enough.

```
private Node<E> toAVLTree(Node<E> localRoot)
{
    if (localRoot == null) {
        // If the tree is empty, then it is balanced.
        return null;
    }
    else
    {
        // If the tree is not empty, then we need to check if it is balanced.
        // Check for the left, then the right subtrees.
        // In each level, find the height of the left and right subtrees.

        localRoot.left = toAVLTree(localRoot.left);
        localRoot.right = toAVLTree(localRoot.right);

        int rightHeight = 0;
        int leftHeight = 0;

        if (localRoot.right != null)
            rightHeight = height(localRoot.right);

        if (localRoot.left != null)
            leftHeight = height(localRoot.left);

        // If the difference is greater than 1, then the tree is not balanced.
        if (Math.abs(rightHeight - leftHeight) > 1) {
            // Once we find that the tree is not balanced, we will set the BALANCED value to false
            // to check the tree again.
            BALANCED = false;
            // If the tree is unbalanced, then we need to find if tree is,
            // left-left, left-right, right-left, or right-right tree.
            int rightChildHeight = 0;
            int leftChildHeight = 0;

            // If tree is right tree.
            if (rightHeight > leftHeight)
            {
                Node<E> rightChild = localRoot.right;
                // Find the height of the right and left child.
                if (rightChild.right != null)
                    rightChildHeight = height(rightChild.right);
                if (rightChild.left != null)
                    leftChildHeight = height(rightChild.left);

                // If tree is right-left tree, then we need to rotate child to the right,
                // and then rotate the parent to the left.
                if (leftChildHeight > rightChildHeight)
                {
                    localRoot.right = rotateRight(rightChild);
                    return rotateLeft(localRoot);
                }
                // If tree is right-right tree, then we just need to rotate the parent to the left.
                else
                {
                    return rotateLeft(localRoot);
                }
            }
```

```
            // If tree is left tree.
            else
            {
                Node<E> leftChild = localRoot.left;
                // Find the height of the right and left child.
                if (leftChild.right != null)
                    rightChildHeight = height(leftChild.right);
                if (leftChild.left != null)
                    leftChildHeight = height(leftChild.left);

                // If tree is left-right tree, then we need to rotate child to the left,
                // and then rotate the parent to the right.
                if (rightChildHeight > leftChildHeight)
                {
                    localRoot.left = rotateLeft(leftChild);
                    return rotateRight(localRoot);
                }
                // If tree is left-left tree, then we just need to rotate the parent to the right.
                else
                {
                    return rotateRight(localRoot);
                }
            }
        }
        // If the tree is balanced, then we will return the root of the tree.
        else
        {
            return localRoot;
        }
    }
}
```

Let’s look at the other helper methods.

`private rotateRight()`

This method takes root of a binary tree and rotates it to the right.

```
protected Node<E> rotateRight(Node<E> root)
{
    Node<E> temp = root.left;
    root.left = temp.right;
    temp.right = root;
    return temp;
}
```

`private rotateLeft()`

This method takes root of a binary tree and rotates it to the left.

```
public Node<E> rotateLeft(Node<E> root)
{
    Node<E> temp = root.right;
    root.right = temp.left;
    temp.left = root;
    return temp;
}
```

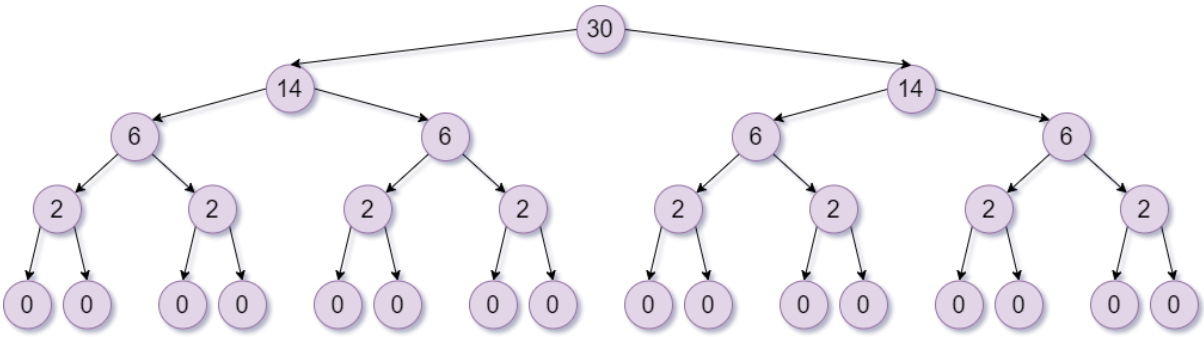
`private height()`

This methods, iterates the node until the reaches the leaves and returns the furthest height.

```
private int height(Node<E> node)
{
    if (node == null)
        return 0;

    // Return 1 + max(the height of the left or right subtrees)
    return (1 + Math.max(height(node.left),
                         height(node.right)));
}
```

Run-time complexity:

`toAVLTree()` method, inside the while loop, invokes the `private toAVLTree()` method until the tree is balanced. And, inside the `private toAVLTree()` calculating height and rotation operations will be performed. To understand the best case of this operation, let's look at the diagram below.



In this diagram, I showed the number of recursive calls for the obtain if the subtrees are left heavy or right heavy tree by calculating heights of both children sizes.
This is a best case for the `toAVLTree()` method, in this case height of each left and right subtrees of all nodes will be same, therefore, the method will not need to obtain if it is left-right or right-left tree neither it will not perform any rotation operation.

Number of recursive calls in the best case:

$$= (2^0 2^h - 2) + (2^1 2^{h-1} - 4) + (2^2 2^{h-2} - 8) + \cdots + (2^h 2^{h-h} - 2^h)$$

$$= (2^h - 2) + (2^h - 4) + (2^h - 8) + \cdots + (2^h - 2^h)$$

$$\sum_{a=0}^{h} \left(-2 \cdot 2^a + 2^h\right) = 2^h (h+1) - 2^{h+2} + 2$$

Result of series is calculated here.
We know h = log(n+1) in a perfect tree where n is number of nodes.
Therefore, the result of series will be as follows:

$$= 2^{log_2(n+1)}.(log_2(n+1)+1) - 2^2.2^{log_2(n+1)} + 2$$

$$= (n+1).(log_2(n+1)+1) - 4(n+1) + 2$$

$$= n\big(log_2(n+1)\big) + log_2(n+1) - 3n - 1$$

Finally, in the best case, run time complexity of `toAVLTree()` method is: *Θ(nlogn)*

Time complexity in general will be: Ω *(nlogn)* (I don't have information about worst case since the tree can be in any shape, theoretically it is hard to calculate)

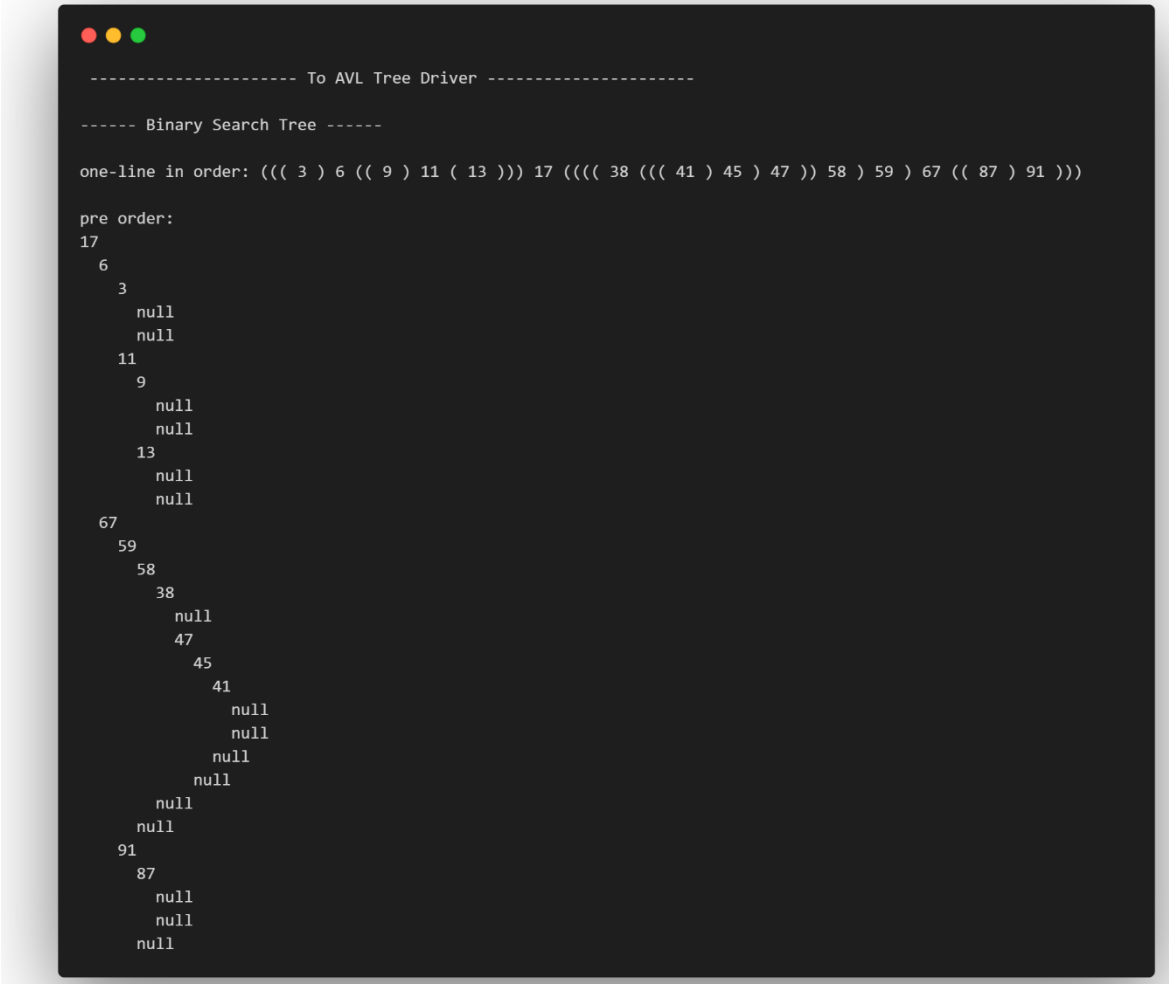
4. Command Results

(Values are generated randomly, result will be change in every run)

1. `toBinarySearchTreeWithArrayElements()`

```
---------------------- To Binary Search Tree With Array Elements Driver ----------------------

----- Binary Tree -----
55
  21
    null
    53
      null
      92
        null
        null
  71
    55
      null
      84
        75
          null
          null
        null
    50
      91
        null
        null
      null

Binary Tree (one-line inorder):
(( 21 ( 53 ( 92 ))) 55 (( 55 (( 75 ) 84 )) 71 (( 91 ) 50 )))


-- The array: [18, 31, 17, 9, 46, 36, 48, 3, 42, 12] --

----- Binary Search Tree -----
17
  3
    null
    9
      null
      12
        null
        null
  42
    18
      null
      36
        31
          null
          null
        null
    48
      46
        null
        null
      null

Binary Search Tree (one-line inorder):
(( 3 ( 9 ( 12 ))) 17 (( 18 (( 31 ) 36 )) 42 (( 46 ) 48 )))
```

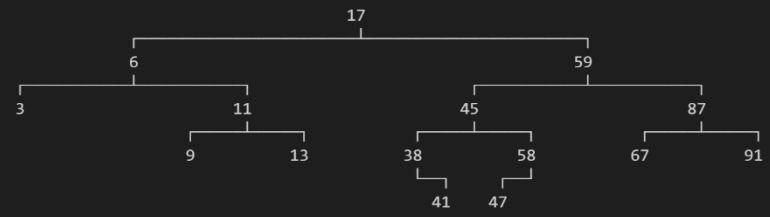
2. toAVLTree()



```
 ---------------------- To AVL Tree Driver ----------------------

------ Binary Search Tree ------

one-line in order: ((( 3 ) 6 (( 9 ) 11 ( 13 ))) 17 (((( 38 ((( 41 ) 45 ) 47 )) 58 ) 59 ) 67 (( 87 ) 91 )))

pre order:
17
  6
    3
      null
      null
    11
      9
        null
        null
      13
        null
        null
  67
    59
      58
        38
          null
          47
            45
              41
                null
                null
              null
            null
        null
      null
    91
      87
        null
        null
      null
```

continues...

```
------ AVL Tree ------
```




```
one-line in order: ((( 3 ) 6 (( 9 ) 11 ( 13 ))) 17 ((( 38 ( 41 )) 45 (( 47 ) 58 )) 59 (( 67 ) 87 ( 91 ))))

pre order:
17
  6
    3
      null
      null
    11
      9
        null
        null
      13
        null
        null
  59
    45
      38
        null
        41
          null
          null
      58
        47
          null
          null
        null
    87
      67
        null
        null
      91
        null
        null
```