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

Student Name: Enes Abdülhalik
Student Number: 1901042803

Question 1:

1- System Requirements:

a. The system has function that creates a binary search tree using the elements of an array with n elements and follows the structure of a binary tree with n elements.
b. Both the binary tree and the array inputs must have the same number of elements.
c. Entering null values or the binary tree and the array with different sizes will lead to an IllegalArgumentException.
d. A stack implementation that is different from the standard java implementation is used and is included in the source code.

2- Problem-Solution Approach:

The system has three functions:

- Public BinarySearchTree<E> toBST(BinaryTree<E> b , E[]);

This is a wielder function. It checks the validity of the input, sorts the array using quick sort, stores the order of insertion of the elements and inserts the elements into a binary search tree.

- Private Stack<E> sortInsert(BinaryTree<E>, E[], Stack<E>):

This is a helper function that creates and returns a stack that contains the elements ordered by their insertion place. It uses a recursive solution to insert the elements into the stack based on their expected place in the given structure.

- Private int BTSize(BinaryTree<E>);

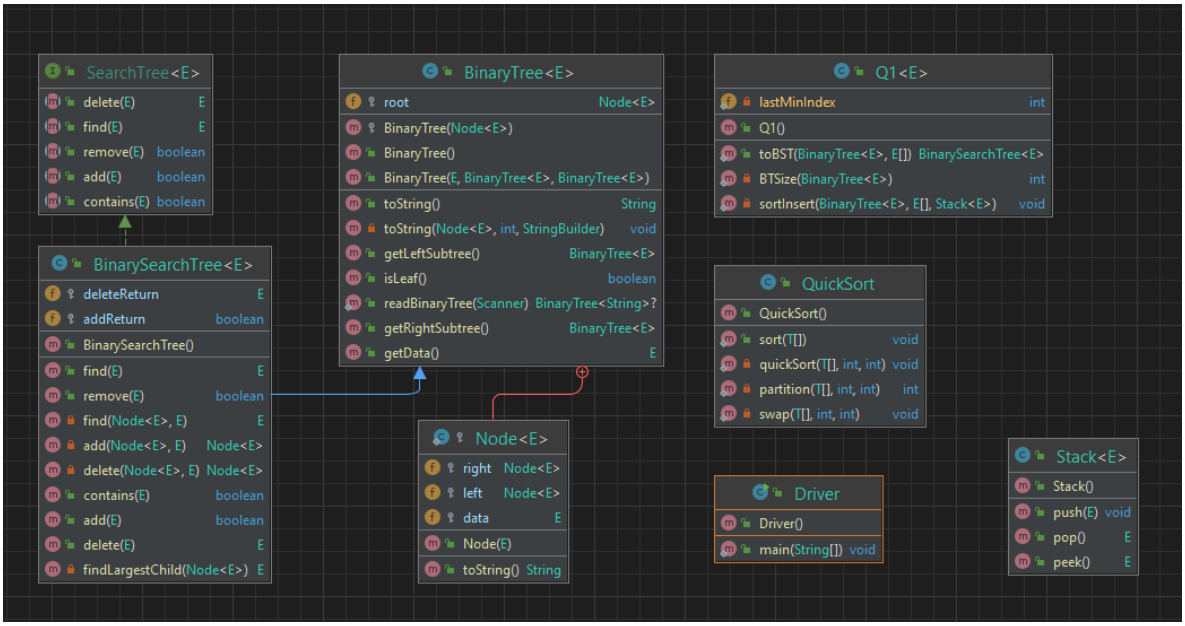
Returns the number of nodes in the binary tree.

And a static data field:

- static int lastMinIndex;

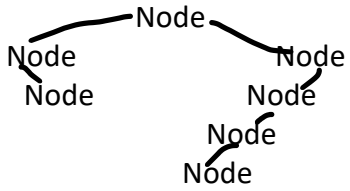
It represents the index of the last element that was inserted into the stack.

3- Class Diagram:



4- Test Cases:

The elements {3, 5, 12, 4, 1, 7, 6}, and a binary tree with the following structure:



Are given to the function and the resulting binary search tree is printed to the terminal.

5- Running and Results:

The driver code is designed to be launched by a makefile in a Linux system.

Result:

```
4
    1
        null
        3
            null
            null
    12
        7
            5
                6
                    null
                    null
                null
            null
        null
```

It matches the given structure.

6- Time Complexity: n = number of elements

```
public static <E extends Comparable<E>> BinarySearchTree<E> toBST(BinaryTree<E> binaryTree, E[]
array) {
    if  (array == null || binaryTree == null ||
        BTSize(binaryTree) != array.length)        Θ(n)
        throw new IllegalArgumentException();

    //Sort the array
    E[] arraySorted = array.clone();    Θ(n)
    QuickSort.sort(arraySorted);        O(nlogn)
    //Get the order of insertion of the elements
    Stack<E> order = new Stack<>();            Θ(1)
```

```
        sortInsert(binaryTree, arraySorted, order);  Θ(n)

        //Add the elements to the tree according to the order
        BinarySearchTree<E> result = new BinarySearchTree<>();          Θ(1)
        while(!order.isEmpty())       Θ(n)
            result.add(order.removeFirst()); Θ(1)

        lastMinIndex = 0;       Θ(1)
        return result;        Θ(1)
}
                                T(n) = O(n)

private static <E extends Comparable<E>> void sortInsert(BinaryTree<E> binaryTree, E[] array,
Stack<E> insertOrder) {
        if (binaryTree == null)     Θ(1)
            return;

        //Get the order of insertion for the left subtree
        sortInsert(binaryTree.getLeftSubtree(), array, insertOrder);  T(n/2)
        //Store the element for the root
        E root = array[lastMinIndex];         Θ(1)
        ++lastMinIndex;               Θ(1)
        //Get the order of insertion for the left subtree and add it to the total
        sortInsert(binaryTree.getRightSubtree(), array, insertOrder);     T(n/2)

        //Insert the element for the root
        insertOrder.push(root);       Θ(1)
}
                    T(n) = 2*T(n/2) + Θ(1)          (Worst case)
                    T(n) = Θ(1)             if n = 0    (base case)
                ➔ T(n) = (2^logn)* Θ(1) = Θ(n)       (General case)

private static <E extends Comparable<E>> int BTSize(BinaryTree<E> binaryTree) {
        if (binaryTree == null)         Θ(1)
            return 0;

        return 1 + BTSize(binaryTree.getLeftSubtree()) +
                    BTSize(binaryTree.getRightSubtree());  2*T(n/2)
}
                    T(n) = 2*T(n/2) + Θ(1)          (Worst case)
                    T(n) = Θ(1)             if n = 0    (base case)
                ➔ T(n) = (2^logn)* Θ(1) = Θ(n)       (General case)
```

Question 2:

1- System Requirements:

a. The system takes a binary search tree and rearranges it and returns it as an AVL tree.
b. The nodes in the returned AVL tree are the same nodes in the given binary search tree, no cloning was done.
c. The class Q2 is included in the trees package because of some access limitations to binary search tree and AVL tree members.

2- Problem-Solution Approach:

The system has five functions:

- Public static <E extends Comparable<E>> AVLTree<E> BSTToAVL(BinarySearchTree<E>);

This is a wielder function. It checks the validity of the input, calls the balance method, and constructs an AVL tree object from the root returned by the balance method.

- Private static <E extends Comparable<E>> AVLNode<E> balance (AVLNode<E>);

This function recursively balances every single node that is under the given root. It uses custom rotation functions provided in the class.

The function keeps count of the root's left and right subtrees' height to calculate its balance accurately in the following form: The balance of a node = right subtree's height - left subtree's height. The height of each subtree is calculated when advancing in that subtree. So, to have the height of the left subtree, the left subtree must be completely sorted, and before it exits its call, its total height is recorded in a static parameter called "height".

In each call, a rotation is done, then a recursive call is done for the subtree in the direction of the rotation to be sure it's in the right form, and then the height record of it is updated.

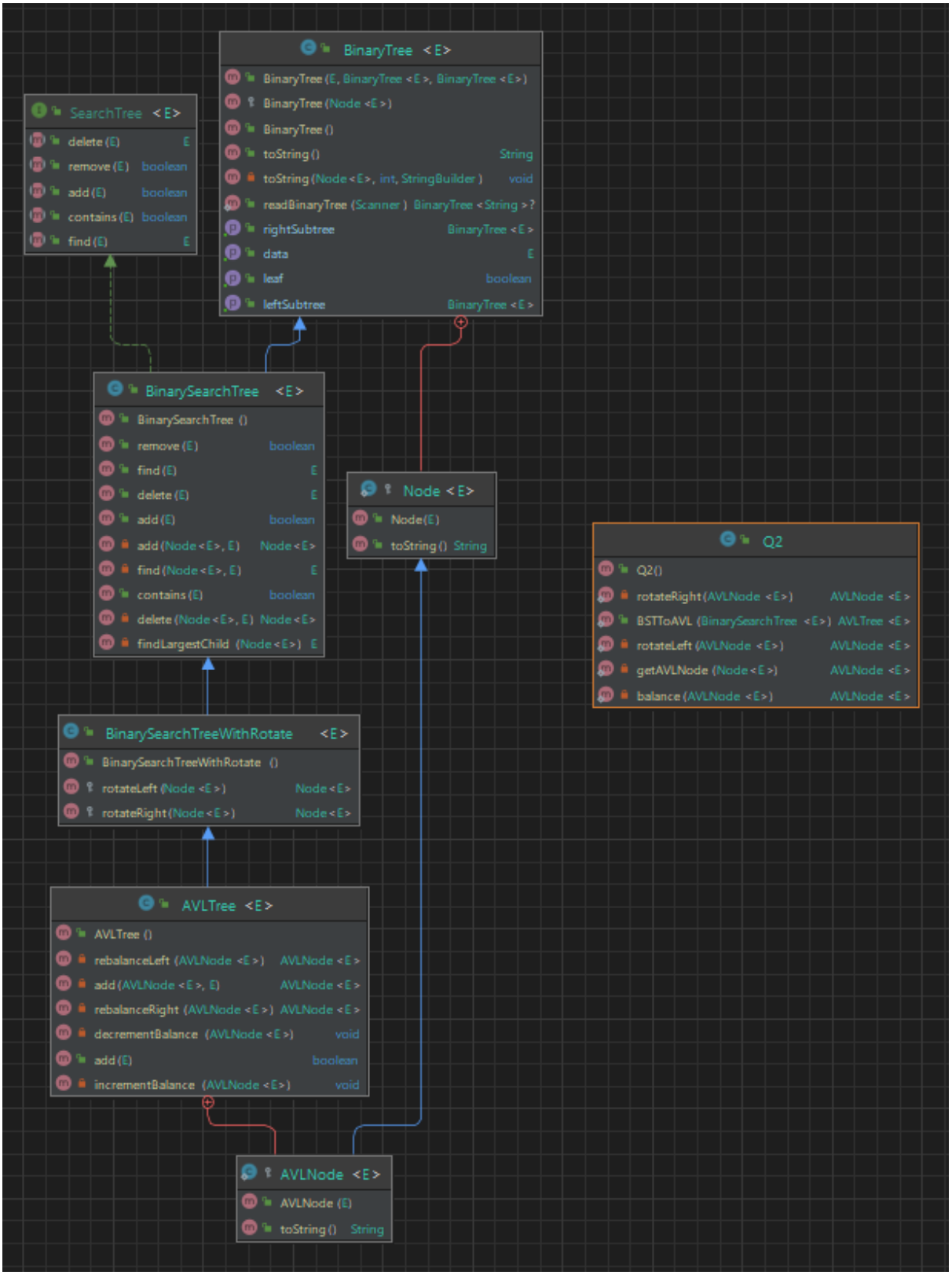
- Private static <E extends Comparable<E>> AVLNode<E> rotateRight(AVLNode<E>);
  And
  Private static <E extends Comparable<E>> AVLNode<E> rotateLeft(AVLNode<E>);

They return the new root of the tree after rotation. They are ordinary rotation functions, much like the ones in the BinarySearchTreeWithRotate class, except that they reset the balance of the old root to 0 before rotation.

- Private static <E extends Comparable<E>> AVLNode<E> getAVLNode(BinaryTree.Node<E>);

Returns an AVLNode version of a given normal binary tree node. It's used to copy the left and right references of the normal node to the AVL node.

3- Class Diagram:



4- Test Cases:

A randomly generated sequence of 7 integer values is inserted into a binary search tree, and it gets printed on the screen. After that, the binary search tree is converted to an AVL tree, and it gets printed on the screen again.

5- Running and Results:

The driver code is designed to be launched by a makefile in a Linux system.

Binary Search Tree:

```
46
    null
    124
        null
        166
            129
                null
                null
            254
                null
                307
                    282
                        null
                        null
                    null
```

Has a height of 6.

AVL Tree:

```
0: 166
    0: 124
        0: 46
            null
            null
        0: 129
            null
            null
    0: 282
        0: 254
            null
            null
        0: 307
            null
            null
```

Has a height of 3.

6- Time Complexity:

n = length of the container

```
public static <E extends Comparable<E>> AVLTree<E> BSTToAVL(BinarySearchTree<E> bst) {
    if (bst == null)        Θ(1)
        return null;
    bst.root = balance(getAVLNode(bst.root));
    AVLTree<E> result = new AVLTree<>();     Θ(1)
    result.root = bst.root;      Θ(1)
    return result;              Θ(1)
}
```

```
private static <E extends Comparable<E>> AVLNode<E> balance(AVLNode<E> root) {
    if (root != null) {      Θ(1)
        //The root is not null
        //Increment the height and store the current height
        ++height;          Θ(1)
        int baseHeight = height;        Θ(1)

        //Balance the left side and store its height, then reset the current height
        AVLNode<E> left = balance(getAVLNode(root.left));      T(n/2)
        int lheight = height;          Θ(1)
        height = baseHeight;        Θ(1)

        //Balance the right side and store its height, then reset the current height
        AVLNode<E> right = balance(getAVLNode(root.right)); T(n/2)
        int rheight = height;          Θ(1)
        height = baseHeight;        Θ(1)

        //Update the balance and the subtrees of the root
        root.left = left;       Θ(1)
        root.right = right;    Θ(1)
        root.balance = rheight - lheight;      Θ(1)

        if (root.balance < -1){       Θ(1)
            //There is a violation on the left side
            if(left.balance > 0) {     Θ(1)
                /* The violation is Left-Right, rotate the left
                 * subtree to the left then update the balances.
                 */
                root.left = rotateLeft(left);              Θ(1)
                --((AVLNode<E>) root.left).balance;     Θ(1)
                --root.balance;            Θ(1)
            }

            /* Rotate the root to the right, then balance
             * the right side and update the heights.
             */
            root = rotateRight(root);            Θ(1)
            root.right = balance((AVLNode<E>) root.right);        T(n/2)
            rheight = height;          Θ(1)
            lheight--;           Θ(1)
        }

        else if (root.balance > 1) {          Θ(1)
            //There is a violation on the right side
            if(right.balance < 0) {            Θ(1)
```

```
                /* The violation is Right-Left, rotate the right
                 * subtree to the right then update the balances.
                 */
                root.right = rotateRight(right);        Θ(1)
                ++((AVLNode<E>) root.right).balance;      Θ(1)
                ++root.balance;              Θ(1)
            }

            /* Rotate the root to the left, then balance
             * the left side and update the heights.
             */
            root = rotateLeft(root);         Θ(1)
            root.left = balance((AVLNode<E>) root.left);      T(n/2)
            lheight = height;         Θ(1)
            rheight--;              Θ(1)
        }

        //Update the roots balance and the current height
        root.balance = rheight - lheight;     Θ(1)
        height = (lheight + rheight + 1)/2;   Θ(1)
    }

    return root;          Θ(1)
}
                            T(n) = 3T(n/2) + Θ(1)           (Worst case)
                            T(n) = Θ(1)           if n = 0    (base case)
                        ➔ T(n) = (3^logn)* Θ(1) = O(3^logn)       (General case)


private static <E extends Comparable<E>> AVLNode<E> rotateRight(AVLNode<E> root) {
    if(root.left != null){     Θ(1)
        root.balance = 0;    Θ(1)
        AVLNode<E> temp = (AVLNode<E>) root.left;     Θ(1)
        root.left = temp.right;     Θ(1)
        temp.right = root;      Θ(1)
        return temp;     Θ(1)
    }

    return root;      Θ(1)
}
                                        T(n) = Θ(1)
```

```
private static <E extends Comparable<E>> AVLNode<E> rotateLeft(AVLNode<E> root) {
    if(root.right != null){         Θ(1)
        root.balance = 0;           Θ(1)
        AVLNode<E> temp = (AVLNode<E>) root.right;          Θ(1)
        root.right = temp.left;     Θ(1)
        temp.left = root;           Θ(1)
        return temp;            Θ(1)
    }

    return root;        Θ(1)
}
                                                T(n) = Θ(1)

private static <E extends Comparable<E>> AVLNode<E> getAVLNode(BinaryTree.Node<E> node) {
    AVLNode<E> avlNode;     Θ(1)
    if(node == null)        Θ(1)
        avlNode = null;
    else {
        avlNode = new AVLNode<>(node.data);     Θ(1)
        avlNode.left = node.left;       Θ(1)
        avlNode.right = node.right;         Θ(1)
    }
    return avlNode;     Θ(1)
}
                                                T(n) = Θ(1)
```