GTU Department of Computer Engineering

CSE 222 / 505 – Spring 2022

Homework 7 - Report

Ahmet USLUOGLU

1801042602

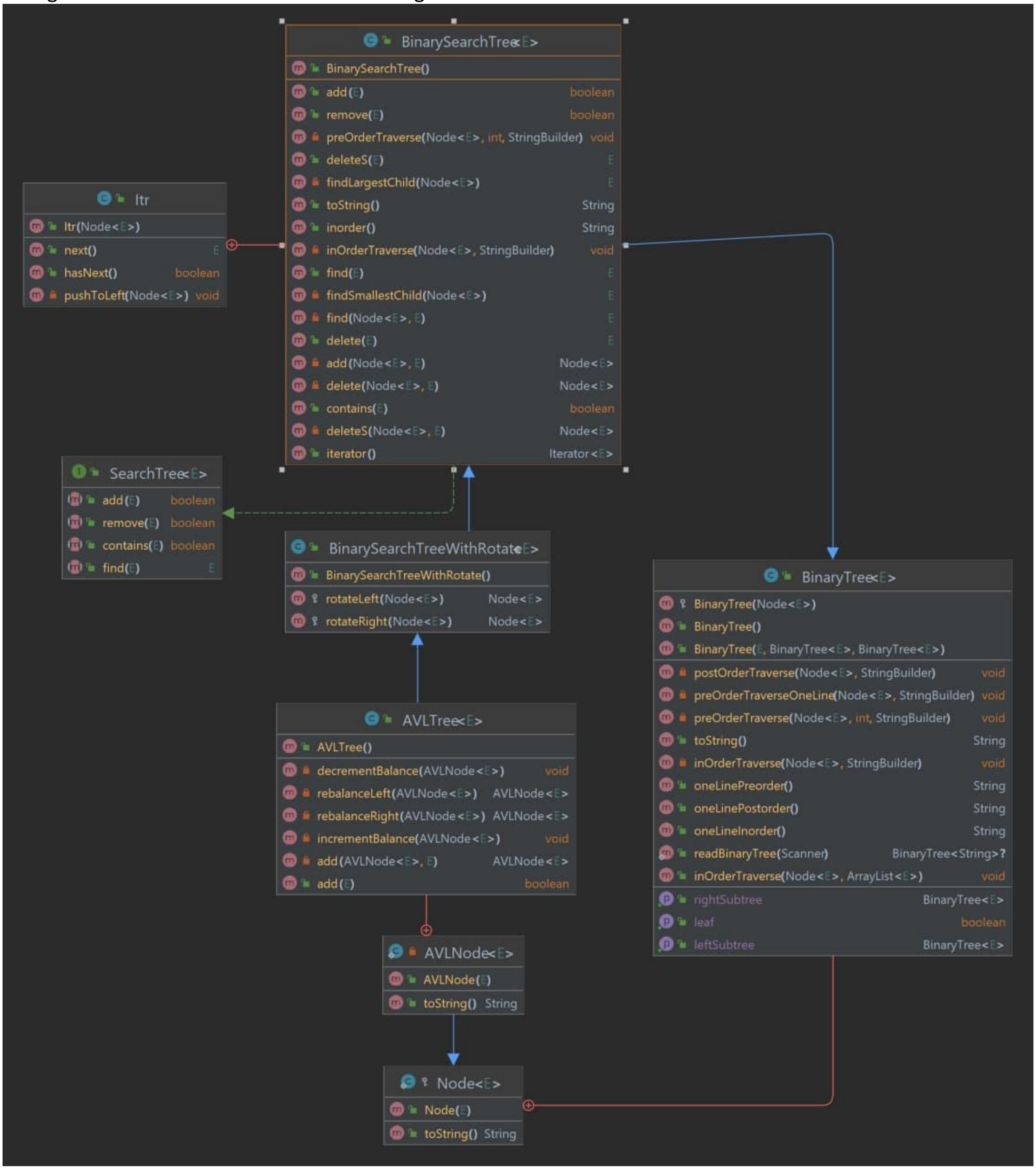
1 – System Requirement

Operating System must have JDK (Java Development Kit) 11 and JRE (Java RuntimeEnvironment) 11 or higher.

There should be enough space for storing data's.

2 – Class Diagrams

*Higher Resolution Version of the Class Diagram is in the files.



3. Problem - Solution Approach

Problem:

1 -) 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 binary tree contains n nodes and doesn’t need to be balanced. 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.

2-) 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.

Solution:

1-) To create a Binary Search Tree with the same structure as Binary Tree using Array elements, I have sorted the array then traversed the Binary Tree with In-Order traversal method. While traversing set the first element in the array to first element found with traversal and removed that element from Array.

2-) To Convert the unbalanced Binary Search Tree to a balanced AVLTree , I have used AVLTree’s rotate left and rotate right functions. After each element is added AVLTree cheks the heightand Balance of the tree and rebalances the tree.

4 – Test Cases

a-) Testing the part1

1 – Adding random elements to the Binary tree and Array.

2 – Sorts the Array

3 – Making a Binary Search Tree with same structure.

b-) Testing the part2

1 -) Make Unbalanced Binary Search Tree

2 -) Making a balanced AVLTree from Unbalanced Binary Search Tree

5 – Running Program and Results

a) Testing the part1

1 – Adding random elements to the Binary tree and Array.

```
Binary Tree's Structure
12
  11
    null
    8
      9
        null
        null
      10
        null
        null
  null


Elements in the Array
[3, 1, 4, 7, 6]
```

2 – Sorts the Array

```
Elements in the Array
[3, 1, 4, 7, 6]

Sorted Array
[1, 3, 4, 6, 7]
```

3 – Making a Binary Search Tree with same structure.

```
Binary Search Tree's Structure with Array's Elements
7
  1
    null
    4
      3
        null
        null
      6
        null
        null
  null
```

b-) Testing the part2

1 -) Make Unbalanced Binary Search Tree.

```
        Unbalanced Binary Search Tree
1
  null
  2
    null
    3
      null
      4
        null
        5
          null
          6
            null
            7
              null
              null
```

2 -) Making a balanced AVLTree from Unbalanced Binary Search Tree

```
        Balanced AVL Tree
4
  2
    1
      null
      null
    3
      null
      null
  6
    5
      null
      null
    7
      null
      null
```

6 – Calculate Time complexity

--Example Part1 complexity

```java
public static <E> void part1(BinaryTree<E> bt, ArrayList<E> al)
    {
        al.sort(null);

        System.out.println("Sorted Array");
        System.out.println(al);
        System.out.println();

        bt.inOrderTraverse(bt.root,al);

        System.out.println("Binary Search Tree's Structure with Array's Elements");
        System.out.println(bt);
        System.out.println();
    }

```

```java
public void inOrderTraverse(Node<E> node, ArrayList<E> al){
        if(node == null){
            //do nothing
        } else {
            inOrderTraverse(node.left, al);
            node.data = al.remove(0);
            inOrderTraverse(node.right, al);
        }
    }
```

- Sorting the array complexity (Merge Sort) = O(nlogn)
- Traversing the Binary Tree takes = O(n)
- Part1 Complexity T(n) = Tetha(nlogn)

--Example Part2 Complexity

```java
private AVLNode<E> add(AVLNode<E> localRoot, E item){
        // If the local root is null, return a new AVLNode with the item inserted
        if(localRoot == null){
            addReturn = true;
            increase = true;
            return new AVLNode<E>(item);
        }

// Compare the item to the data in the current root. If equal, do not insert item
        int compare = item.compareTo(localRoot.data);
        if(compare == 0){
            //Item is already in tree
            increase = false;
            addReturn = false;
            return localRoot;
        }

// If the item is less than the local root's value, recursively call add on left subtree
        if(compare < 0){
            localRoot.left = add((AVLNode<E>) localRoot.left, item);
            if(increase){
                decrementBalance(localRoot);
                if(localRoot.balance < AVLNode.LEFT_HEAVY){
                    increase = false;
                    return rebalanceLeft(localRoot);
                }
            }
            return localRoot; // Re-balance not needed
        }
        else {
            localRoot.right = add((AVLNode<E>) localRoot.right, item);
            if(increase){
                incrementBalance(localRoot);
                if(localRoot.balance > AVLNode.RIGHT_HEAVY){
                    increase = false;
                    return rebalanceRight(localRoot);
                }
            }
            return localRoot; // Re-balance not needed
        }
    }
```

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

```java
private AVLNode<E> rebalanceLeft(AVLNode<E> localRoot){
        //Obtain reference to left child
        AVLNode<E> leftChild = (AVLNode<E>) localRoot.left;
        //see whether left-right heavy
        if(leftChild.balance > AVLNode.BALANCED){
            //Obtain reference to left-right child.
            AVLNode<E> leftRightChild = (AVLNode<E>) leftChild.right;
            /*
             * Adjust the balances to be their new values after the rotations are performed
             */
            if(leftRightChild.balance < AVLNode.BALANCED){
                leftChild.balance = AVLNode.BALANCED;
                leftRightChild.balance = AVLNode.BALANCED;
                localRoot.balance = AVLNode.RIGHT_HEAVY;
            } else if (leftRightChild.balance > AVLNode.BALANCED){
                leftChild.balance = AVLNode.LEFT_HEAVY;
                leftRightChild.balance = AVLNode.BALANCED;
                localRoot.balance = AVLNode.BALANCED;
            } else {
                //Left-Right balanced case
                leftChild.balance = AVLNode.BALANCED;
                localRoot.balance = AVLNode.BALANCED;
            }
            //Perform left rotation
            localRoot.left = rotateLeft(leftChild);
        } else {
            //Left-left case
            leftChild.balance = AVLNode.BALANCED;
            localRoot.balance = AVLNode.BALANCED;
        }
        //now rotate the local root right
        return (AVLNode<E>) rotateRight(localRoot);
    }
```

Traversing Binary Search Tree Complexity = O(n)
AVLTree insertion Complexity = O(logn)
RotateLeft, RotateRight complexity = O(1)
RebalanceLeft, RebalanceRight complexity = O(1)

Part2 complexity = Tetha (nlogn)