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

Sefa Çiçek
1801042657

1. **System Requirements**

- jdk and jre are requested from operating system to execute this java program.

-User has to run makefile for folder

2. Problem Solution Approach

Q1:
Firstly, I checked the array size. If it is 0, the function throws an exception. Otherwise, I sorted the array. I traversed the tree in-order for replacing the binary tree nodes with the array elements. Lastly, I called ‘addNodesToBST’ function to add nodes in binary tree to the binary search tree.

Q2:
I’ve used a wrapper function to implement the question. I called ‘convertToAvl’ function with three parameters. First two parameters are same, they are ‘binary search tree’, last parameter is the ‘root value of the bst’. First two parameter were used for different purposes. First one stands for subtrees on recursive calls, second one stands for exact binary search tree (result of question).
I traversed the BST post-order. After calling left and right subtree, I calculated the ‘balance value’ of the root node with subtracting the height of the right sub-tree from the left sub-tree. Depends on the balance value, I called ‘rightRotate’ or ‘leftRotate’ function. In those functions, I rotated the tree and returned the new root of tree. **After rotation functions, I bound the parent node of the old root with the new root.** For this process, I wrote ‘findParentNodeVal’ function. It returns the value of the parent node, with that value, I found the node and I bound the node with the new root.

3. Test Cases

Q1

- Empty array
- Only 1 element in the array
- Many elements in the array
- Degenerate Binary Tree

Q2

- Basic rotations -> LL, LR, RR, RL rotation with 3 element
- Already balanced tree
- Complicated trees

4. Running Command and Results

Q1

- Empty Array

```
BINARY TREE
-----------
3
  2
    1
      null
      null
    null
  4
    null
    5
      null
      null

BINARY SEARCH TREE
------------------
Array :[null, null, null, null, null]

java.lang.NullPointerException
        at BtToBst.convertToBst(BtToBst.java:24)
        at Main.main(Main.java:127)
```

- Only 1 element in the array

```
BINARY TREE
-----------
4
  null
  null

BINARY SEARCH TREE
------------------
Array :[14]

14
  null
  null
```

- Many elements in the array

```
BINARY TREE
-----------
5
  10
    15
      null
      null
    null
  20
    null
    25
      null
      null

BINARY SEARCH TREE
------------------
Array :[5, 4, 3, 2, 1]

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

```
BINARY TREE
-----------
5
  10
    15
      null
      null
    20
      25
        null
        null
      30
        null
        null
  35
    null
    40
      50
        null
        null
      null

BINARY SEARCH TREE
------------------
Array : [9, 8, 7, 6, 5, 4, 3, 2, 1]

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

- Degenerate Binary Tree

```
BINARY TREE
-----------
5
  null
  10
    null
    15
      null
      20
        null
        25
          null
          30
            null
            35
              null
              40
                null
                45
                  null
                  50
                    null
                    55
                      null
                      60
                        null
                        65
                          null
                          70
                            null
                            75
                              null
                              80
                                null
                                85
                                  null
                                  90
                                    null
                                    95
                                      null
                                      100
                                        null
                                        null
```

```
BINARY SEARCH TREE
------------------
Array :[12, 3, 98, 51, 45, 35, 18, 2, 50, 75, 61, 58, 14, 24, 88, 5, 15, 1, 200, 29]

1
  null
  2
    null
    3
      null
      5
        null
        12
          null
          14
            null
            15
              null
              18
                null
                24
                  null
                  29
                    null
                    35
                      null
                      45
                        null
                        50
                          null
                          51
                            null
                            58
                              null
                              61
                                null
                                75
                                  null
                                  88
                                    null
                                    98
                                      null
                                      200
                                        null
                                        null
```

Q2

- Basic rotations -> LL, LR, RR, RL rotation with 3 element

```
BINARY SEARCH TREE
------------------
30
  20
    10
      null
      null
    null
  null

AVL TREE
--------
20
  10
    null
    null
  30
    null
    null

BINARY SEARCH TREE
------------------
10
  null
  20
    null
    30
      null
      null

AVL TREE
--------
20
  10
    null
    null
  30
    null
    null
```

```
BINARY SEARCH TREE
------------------
10
  null
  30
    20
      null
      null
    null

AVL TREE
--------
20
  10
    null
    null
  30
    null
    null

BINARY SEARCH TREE
------------------
30
  10
    null
    20
      null
      null
  null

AVL TREE
--------
20
  10
    null
    null
  30
    null
    null
```

- Already balanced tree

```
BINARY SEARCH TREE
------------------
10
  5
    null
    null
  20
    null
    null

AVL TREE
--------
10
  5
    null
    null
  20
    null
    null
```

- Complicated trees

```
BINARY SEARCH TREE
------------------
10
  5
    2
      null
      null
    8
      7
        6
          null
          null
        null
      null
  16
    null
    20
      18
        null
        null
      null

AVL TREE
--------
10
  5
    2
      null
      null
    7
      6
        null
        null
      8
        null
        null
  18
    16
      null
      null
    20
      null
      null
```

```
BINARY SEARCH TREE
------------------
50
  30
    10
      5
        null
        null
      15
        null
        null
    null
  70
    null
    90
      null
      110
        null
        null

AVL TREE
--------
50
  10
    5
      null
      null
    30
      15
        null
        null
      null
  90
    70
      null
      null
    110
      null
      null
```

Time Complexities

Q1

```java
int index = 0;  // array index indicator

/**
 * It takes a binary tree and an array of items as input, and it returns
 * a binary search tree (BST) with the same structure of binary tree and
 * items of the array as output.
 *
 * @param <T>
 * @param bTree the binary tree
 * @param arr   the array
 * @return      the generated binary search tree
 * @throws NullPointerException if array is empty
 */
public <T extends Comparable<T>> BinaryTree<T> convertToBst(BinaryTree<T> bTree, T[] arr) throws NullPointerException{

    if(arr.length == 0 || arr[0] == null)
        throw new NullPointerException();

    if (bTree == null)
        return null;

    BinarySearchTree<T> bSearchTree = new BinarySearchTree<>();
    Arrays.sort(arr);    // sorting the array

    /* in-order traversing */
    convertToBst(bTree.getLeftSubtree(), arr);
    bTree.root.data = arr[index++];
    convertToBst(bTree.getRightSubtree(), arr);

    /* adding nodes to binary search tree */
    addNodesToBST(bTree.root, bSearchTree);

    return bSearchTree;
}
```

$\Theta(1)$

$\Theta(1)$

$\rightarrow \Theta(1)$

$\rightarrow O(n.\log(n))$

$\rightarrow T(n/2)$

$\rightarrow \Theta(1)$

$\rightarrow T(n/2)$

$\rightarrow O(n\log(n))$

$T(n) = 2T(n/2) + n\log(n)$

$a=2$, $b=2$, $k=1$, $p=1$

$T(n) = O(n^{\log_2 2} . \log^{1+1} n)$

$T(n) = \underline{O(n.\log^2 n)}$

```java
/**
 * Adds nodes of binary tree to the binary search tree.
 *
 * @param <T>
 * @param root        the root of the binary tree
 * @param bSearchTree the binary search tree
 */
private <T extends Comparable<T>> void addNodesToBST(BinaryTree.Node<T> root, BinarySearchTree<T> bSearchTree) {

    /* base case */
    if (root == null)
        return;

    bSearchTree.add(root.data);

    /* traversing the binary tree nodes */
    addNodesToBST(root.left, bSearchTree);
    addNodesToBST(root.right, bSearchTree);
}
```

$\Theta(1)$

$\rightarrow O(n)$

$2T(n/2)$

$T(n) = 2T(n/2) + n$

$\vdots$

$T(n) = 2^k T(n/2) + kn = 2^k T(1) + kn = \underbrace{n + n\log(n)}$

$n = 2^k$

$k = \log(n)$

$\underline{O(n\log(n))}$

Q2

```java
/**
 * It is wrapper function for converting the binary search tree to AVL tree
 *
 * @param <T>
 * @param bstree the binary search tree to be converted
 * @return the balanced binary search tree (AVL tree)
 */
public <T extends Comparable<T>> BinarySearchTree<T> wrapperConvertToAvl(BinarySearchTree<T> bstree) {

    convertToAvl(bstree, bstree, bstree.root.data);
    return bstree;

}
```

$\rightarrow O(n\log(n))$

```java
/**
 * First and second parameter are the binary search tree. One is used for
 * recursive calls (sub-trees) and the other is used to find the parent of node
 * on the real binary tree. Last parameter is used to determine root of the
 * subtree
 * is the root of the exact binary tree or not.
 *
 * @param <T>
 * @param bstree  the binary search tree - sub-trees -
 * @param realBST the real binary search tree
 * @param rootVal the value of the root of binary search tree
 */
private <T extends Comparable<T>> void convertToAvl(BinarySearchTree<T> bstree, BinarySearchTree<T> realBST,
        T rootVal) {

    /* base case */
    if (bstree == null)
        return;

    /* post-order */
    convertToAvl(bstree.getLeftSubtree(), realBST, rootVal);
    convertToAvl(bstree.getRightSubtree(), realBST, rootVal);

    Integer balance = findBalance(bstree.root);

    // Left Left Case
    if (balance > 1 && findBalance(bstree.root.left) >= 0) {

        T tmpRoot = bstree.root.data; // root of the sub-tree
        bstree.root = rightRotate(bstree.root); // rotate the sub-tree

        /*
         * If the node is not the root of the exact tree,
         * parent of the node must point to the new root of the subtree
         */
        if (tmpRoot != rootVal) {
            List<Integer> result = new ArrayList<>();
            findParentNodeVal(realBST.root, (Integer) tmpRoot, -1, result); // finds parent node value
            int value = result.get(index: 0);
            /*
             * If the value of the parent node is less than the old root data,
             * parent node must point to the right of itself, otherwise left.
             */
            if ((Integer) realBST.findTarget(value).data < (Integer) bstree.root.data)
                realBST.findTarget(value).right = bstree.root;
            else
                realBST.findTarget(value).left = bstree.root;
        }
    }
```

Two recursive calls: $2T(n/2)$

findBalance(bstree.root): $\rightarrow O(n)$

findBalance(bstree.root.left): $O(n)$

tmpRoot / rightRotate: $\Theta(1)$

findParentNodeVal: $\rightarrow O(n)$

findTarget: $O(n)$

if / else with findTarget: $O(n)$

if (tmpRoot != rootVal) block: $O(n)$

Left Left Case block: $O(n)$

$$T(n) = 2T(n/2) + n \qquad \boxed{O(n\log(n))}$$

$$T(n) = 2^k T(n/2^k) + kn = 2^k \underbrace{T(1)}_{} + kn$$

$\underbrace{T(n/2^k)}_{T(1)}$

$n = 2^k$

$k = \log(n)$

$2^k \rightarrow n$, $k \rightarrow \log(n)$

$n + \log(n)$

```
// Right Right Case
if (balance < -1 && findBalance(bstree.root.right) <= 0) {
    T tmpRoot = bstree.root.data;
    bstree.root = leftRotate(bstree.root);
    if (tmpRoot != rootVal) {
        List<Integer> result = new ArrayList<>();
        findParentNodeVal(realBST.root, (Integer) tmpRoot, -1, result);
        int value = result.get(index: 0);
        if ((Integer) realBST.findTarget(value).data < (Integer) bstree.root.data)
            realBST.findTarget(value).right = bstree.root;
        else
            realBST.findTarget(value).left = bstree.root;
    }
}
```

O(n)

```
// Left Right Case
if (balance > 1 && findBalance(bstree.root.left) == -1) {
    T tmpRoot = bstree.root.data;
    bstree.root.left = leftRotate(bstree.root.left);
    bstree.root = rightRotate(bstree.root);
    if (tmpRoot != rootVal) {
        List<Integer> result = new ArrayList<>();
        findParentNodeVal(realBST.root, (Integer) tmpRoot, -1, result);
        int value = result.get(index: 0);
        if ((Integer) realBST.findTarget(value).data < (Integer) bstree.root.data)
            realBST.findTarget(value).right = bstree.root;
        else
            realBST.findTarget(value).left = bstree.root;
    }
}
```

O(n)

```
// Right Left Case
if (balance < -1 && findBalance(bstree.root.right) == 1) {
    T tmpRoot = bstree.root.data;
    bstree.root.right = rightRotate(bstree.root.right);
    bstree.root = leftRotate(bstree.root);
    if (tmpRoot != rootVal) {
        List<Integer> result = new ArrayList<>();
        findParentNodeVal(realBST.root, (Integer) tmpRoot, -1, result);
        int value = result.get(index: 0);
        if ((Integer) realBST.findTarget(value).data < (Integer) bstree.root.data)
            realBST.findTarget(value).right = bstree.root;
        else
            realBST.findTarget(value).left = bstree.root;
    }

}
```

O(n)

```
/**
 * Starter method findTarget.
 *
 * @param target The Comparable object being sought
 * @return The object, if found, otherwise null
 */
public Node<E> findTarget(int target) {
  return findTarget(root, target);
}

/**
 * Recursive find target method.
 *
 * @param localRoot The local root
 * @param target    The object being sought
 * @return The object, if found, otherwise null
 */
private Node<E> findTarget(Node<E> localRoot, int target) {
  if (localRoot == null)
    return null;

  // Compare the target with the data field at the root.
  if (target == (Integer) localRoot.data)
    return localRoot;
  else if (target < (Integer) localRoot.data)
    return findTarget(localRoot.left, target);
  else
    return findTarget(localRoot.right, target);
}
```

General case → O(n)

worst case → skewed tree ⇒ O(n)

Best case → T(n) = T(n/2) ⇒ O(logn)

O(1)

→ T(n/2)

→ T(n/2)

```java
/**
 * It finds the parent node value of the node with the given node value.
 *
 * @param <T>
 * @param node   the root of the tree
 * @param value  the node value to be find its parent
 * @param parent the value of the parent
 * @param result the result includes parent value
 */
private <T extends Comparable<T>> void findParentNodeVal(BinaryTree.Node<T> node, Integer value, Integer parent,
        List<Integer> result) {
    if (node == null)
        return;
    if ((Integer) node.data == value) {
        result.add((Integer) parent);
    } else {
        findParentNodeVal(node.left, value, (Integer) node.data, result);
        findParentNodeVal(node.right, value, (Integer) node.data, result);
    }
}
```

$\Theta(1)$

$\rightarrow O(1)$

$2T(n/2)$

$T(n) = 2T(n/2) = \underline{O(n)}$

```java
/**
 * Rotates the tree to the right for balancing. It returns the new root of the
 * sub-tree.
 *
 * @param <T>
 * @param node the root of the sub-tree
 * @return the new root
 */
private <T extends Comparable<T>> BinarySearchTree.Node<T> rightRotate(BinarySearchTree.Node<T> node) {

    BinarySearchTree.Node<T> root = node.left;

    node.left = root.right;
    root.right = node;

    return root;

}
```

$\rightarrow \Theta(1)$

$\Theta(1)$

$\underline{\Theta(1)}$

```java
/**
 * Rotates the tree to the left for balancing. It returns the new root of the
 * sub-tree.
 *
 * @param <T>
 * @param node the root of the sub-tree
 * @return the new root
 */
private <T extends Comparable<T>> BinarySearchTree.Node<T> leftRotate(BinarySearchTree.Node<T> node) {

    BinarySearchTree.Node<T> root = node.right;

    node.right = root.left;
    root.left = node;

    return root;
}
```

$\rightarrow \Theta(1)$

$\Theta(1)$

$\underline{\Theta(1)}$

```java
/**
 * Finds the height of the node with recursive calls for left and right subtree.
 *
 * @param <T>
 * @param node the node to be found it's height
 * @return the height of the node
 */
private <T extends Comparable<T>> Integer findHeight(BinarySearchTree.Node<T> node) {
    if (node == null)
        return -1;
    if (isLeaf(node))
        return 0;
    return 1 + Math.max(findHeight(node.left), findHeight(node.right));
}
```

$\Theta(1)$

$T(n) = 2T(n/2) + 1 \Rightarrow O(n)$

$2T(n/2)$

```java
/**
 * Subtracts the height of the right node of the node from left node.
 *
 * @param <T>
 * @param node the node to be found it's balance value
 * @return the balance value
 */
private <T extends Comparable<T>> Integer findBalance(BinarySearchTree.Node<T> node) {
    if (node == null)
        return 0;

    return findHeight(node.left) - findHeight(node.right);
}
```

$\Theta(1)$

$O(n)$ $O(n)$

$O(n)$

```java
/**
 * Finds whether the node is leaf or not.
 *
 * @param <T>
 * @param node the node of the tree
 * @return true if node is leaf, otherwise false
 */
private <T extends Comparable<T>> boolean isLeaf(BinarySearchTree.Node<T> node) {
    return (node.left == null && node.right == null);
}
```

$\rightarrow \Theta(1)$