CSE 222

HOMEWORK 7 REPORT

ERSEL CELAL EREN

1901042673

1 - System Requirements

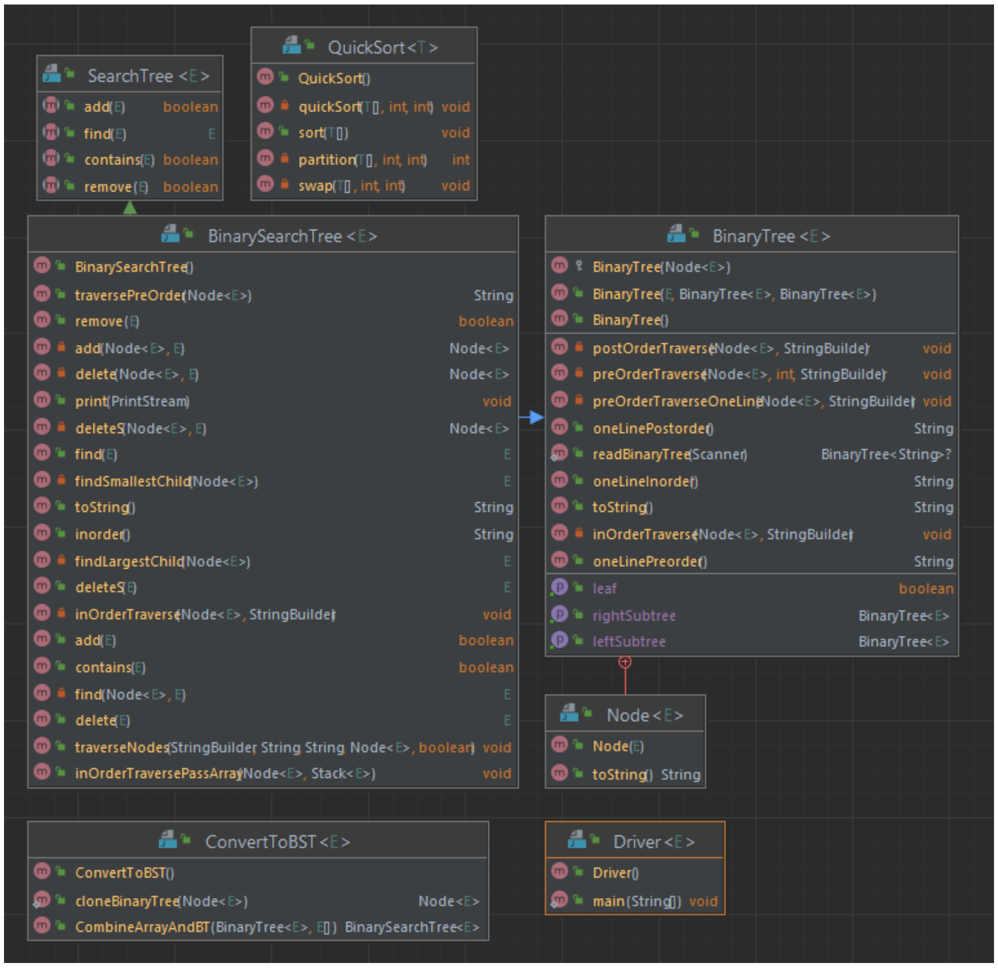
In this homework, we are asked to implement a method that takes Binary Tree and an array and returns Binary Search Tree in the structure of Binary Tree after combined with array.

Second task is that implement a method taking Binary Search Tree as parameter and converting it to AVL tree by using rotations.

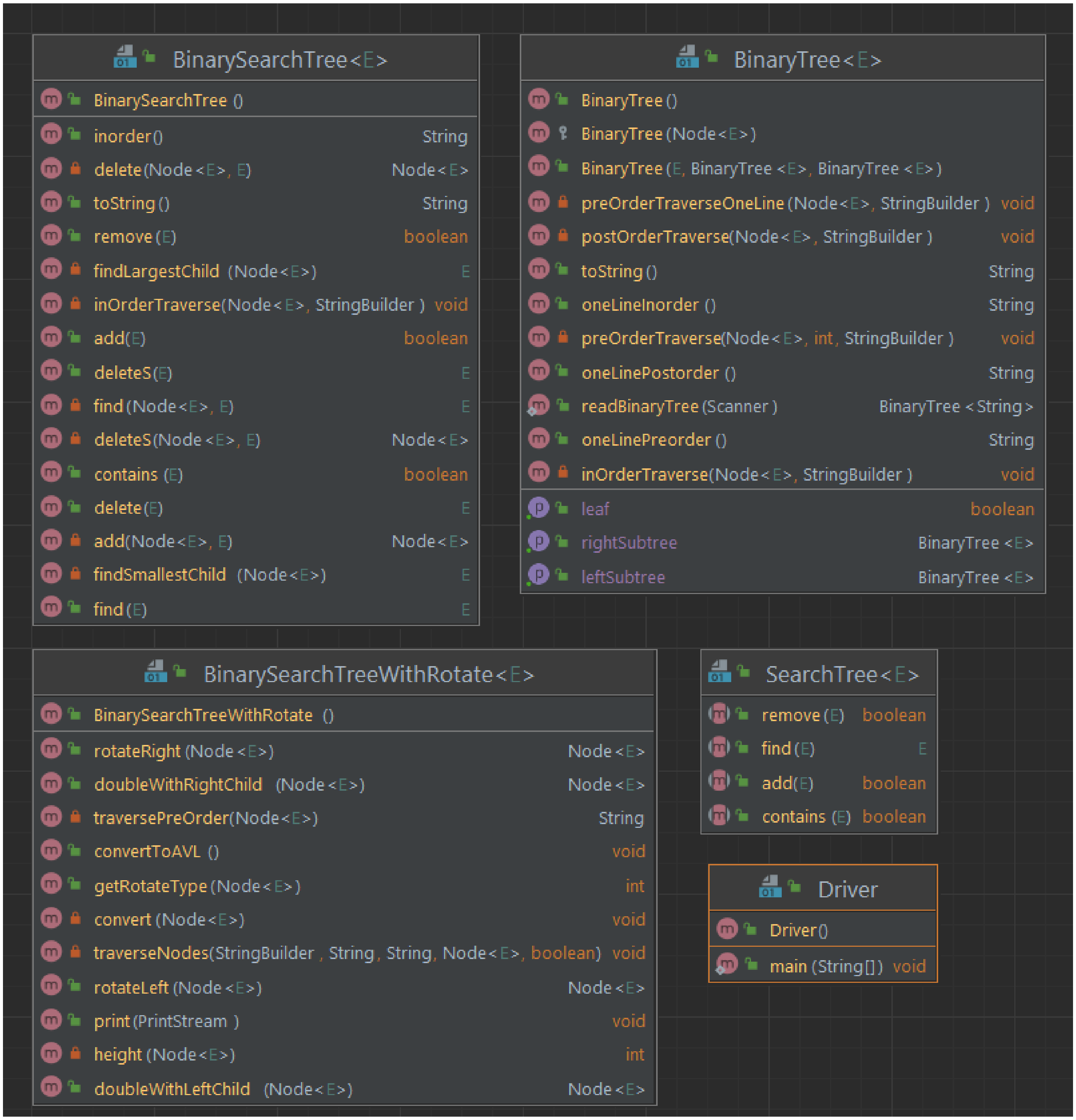
For both two question, there is two different file and Driver code. Methods and classes doesn't need user and there is no menu. Driver classes have test cases.

2 – Class Diagrams

Q1-Class Diagrams


SearchTree <E>
add(E) boolean
find(E) E
contains(E) boolean
remove(E) boolean
QuickSort<T>
QuickSort()
quickSort(T[], int, int) void
sort(T[]) void
partition(T[], int, int) int
swap(T[], int, int) void
BinarySearchTree <E>
BinarySearchTree()
traversePreOrder(Node<E>) String
remove(E) boolean
add(Node<E>, E) Node<E>
delete(Node<E>, E) Node<E>
print(PrintStream) void
deleteS(Node<E>, E) Node<E>
find(E) E
findSmallestChild(Node<E>) E
toString() String
inorder() String
findLargestChild(Node<E>) E
deleteS(E) E
inOrderTraverse(Node<E>, StringBuilder) void
add(E) boolean
contains(E) boolean
find(Node<E>, E) E
delete(E) E
traverseNodes(StringBuilder, String, String, Node<E>, boolean) void
inOrderTraversePassArray(Node<E>, Stack<E>) void
BinaryTree <E>
BinaryTree(Node<E>)
BinaryTree(E, BinaryTree<E>, BinaryTree<E>)
BinaryTree()
postOrderTraverse(Node<E>, StringBuilder) void
preOrderTraverse(Node<E>, int, StringBuilder) void
preOrderTraverseOneLine(Node<E>, StringBuilder) void
oneLinePostorder() String
readBinaryTree(Scanner) BinaryTree<String>?
oneLineInorder() String
toString() String
inOrderTraverse(Node<E>, StringBuilder) void
oneLinePreorder() String
leaf boolean
rightSubtree BinaryTree<E>
leftSubtree BinaryTree<E>
Node <E>
Node(E)
toString() String
ConvertToBST <E>
ConvertToBST()
cloneBinaryTree(Node<E>) Node<E>
CombineArrayAndBT(BinaryTree<E>, E[]) BinarySearchTree<E>
Driver <E>
Driver()
main(String[]) void

Q2-Class Diagrams



BinarySearchTree<E>

BinarySearchTree ()	
inorder()	String
delete(Node<E>, E)	Node<E>
toString ()	String
remove (E)	boolean
findLargestChild (Node<E>)	E
inOrderTraverse(Node<E>, StringBuilder)	void
add(E)	boolean
deleteS(E)	E
find (Node<E>, E)	E
deleteS(Node<E>, E)	Node<E>
contains (E)	boolean
delete(E)	E
add(Node<E>, E)	Node<E>
findSmallestChild (Node<E>)	E
find (E)	E

BinaryTree<E>

BinaryTree ()	
BinaryTree (Node<E>)	
BinaryTree (E, BinaryTree <E>, BinaryTree <E>)	
preOrderTraverseOneLine (Node<E>, StringBuilder)	void
postOrderTraverse(Node<E>, StringBuilder)	void
toString ()	String
oneLineInorder ()	String
preOrderTraverse(Node<E>, int, StringBuilder)	void
oneLinePostorder ()	String
readBinaryTree (Scanner)	BinaryTree <String>
oneLinePreorder ()	String
inOrderTraverse(Node<E>, StringBuilder)	void
leaf	boolean
rightSubtree	BinaryTree <E>
leftSubtree	BinaryTree <E>

BinarySearchTreeWithRotate<E>

BinarySearchTreeWithRotate ()	
rotateRight (Node<E>)	Node<E>
doubleWithRightChild (Node<E>)	Node<E>
traversePreOrder(Node<E>)	String
convertToAVL ()	void
getRotateType (Node<E>)	int
convert (Node<E>)	void
traverseNodes(StringBuilder , String, String, Node<E>, boolean)	void
rotateLeft (Node<E>)	Node<E>
print (PrintStream)	void
height (Node<E>)	int
doubleWithLeftChild (Node<E>)	Node<E>

SearchTree<E>

remove (E)	boolean
find (E)	E
add(E)	boolean
contains (E)	boolean

Driver

Driver()	
main (String[])	void

3 – Problem Solution Approach

For first question, I made root,right and left field of Binary Tree public to pass BT to method. Then I implemented a class named ConvertToBST to convert it to BST. In this class, first I cloned structure of BT into BST. Then I sorted array and pushed it into stack. Finally, I passed array to BST in inOrderTraverse function one by one.

For second question, I implemented BinarySearchTreeWithRotate class. It has rotate,height and convert methods. Convert method starts from the root and traverse tree as preorder. At each traverse, calculates left and right subtree heights and decides rotate type. Rotate types are left-left, left-right, right-right, right-left. Left-left and right-right has one rotation, left-right and right-left types need double rotation.

4 – TEST CASES

Q1 – Converting from BinaryTree to BinarySearchTree

Case 1 : This array will be replaced by BinaryTree.

```
myArray1[0] = 30;
myArray1[1] = 4;
myArray1[2] = 100;
myArray1[3] = 72;
myArray1[4] = 14;
myArray1[5] = 10;
```

Q2 - For Converting from BinarySearchTree to AVL Tree

Case1 : This is case for worst type of BinarySearchTree. To convert this tree all rotate types will be left. So left rotate is tested.

```
bst2.add( 6 );
bst2.add( 5 );
bst2.add( 4 );
bst2.add( 3 );
bst2.add( 2 );
bst2.add( 1 );
```

Case 2: For this case, both left rotate and double rotates are used.

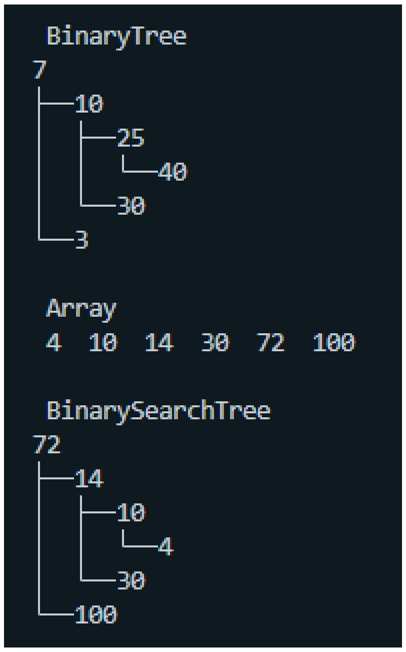
```
        bst.add( 100);
        bst.add( 150);
        bst.add( 50);
        bst.add( 10);
        bst.add( 5);
        bst.add( 60);
        bst.add( 20);
        bst.add( 3);
        bst.add( 6);
        bst.add( 25);
        bst.add( 30);
        bst.add( 2);
        bst.add( 0);
        bst.add( 150);
        bst.add( 120);
        bst.add( 110);
        bst.add( 130);
        bst.add( 250);
```

Case 3 : In this case right rotates are used.

```
        bst3.add( 100);
        bst3.add( 200);
        bst3.add( 300);
        bst3.add( 150);
        bst3.add( 125);
        bst3.add( 110);
        bst3.add( 105);
        bst3.add( 400);
```

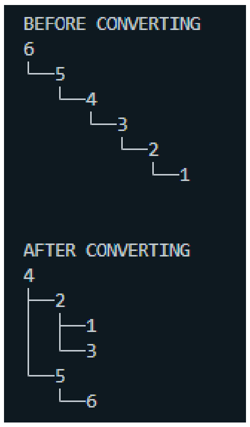
5 – RESULTS

Q1 – Test Results

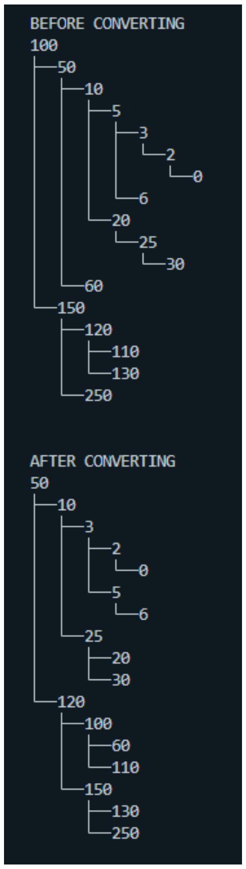


```
 BinaryTree
7
├──10
│   ├──25
│   │   └──40
│   └──30
└──3

 Array
 4  10  14  30  72  100

 BinarySearchTree
72
├──14
│   ├──10
│   │   └──4
│   └──30
└──100
```

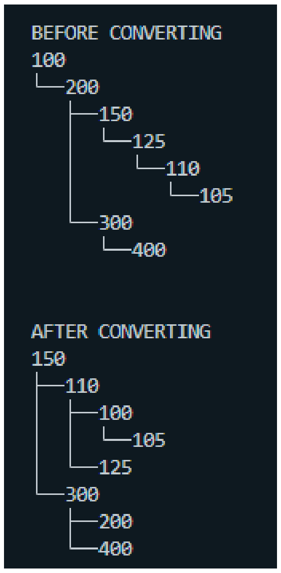
2 – Results



```
BEFORE CONVERTING
6
└──5
    └──4
        └──3
            └──2
                └──1

AFTER CONVERTING
4
├──2
│   ├──1
│   └──3
└──5
    └──6
```



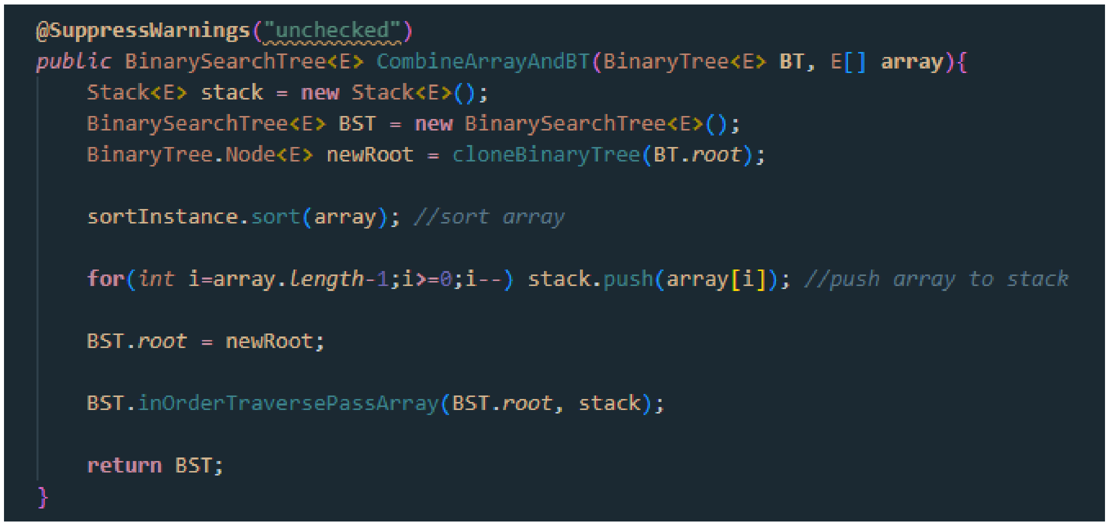
```
BEFORE CONVERTING
100
├──50
│   ├──10
│   │   ├──5
│   │   │   ├──3
│   │   │   │   └──2
│   │   │   │       └──0
│   │   │   └──6
│   │   └──20
│   │       └──25
│   │           └──30
│   └──60
└──150
    ├──120
    │   ├──110
    │   └──130
    └──250

AFTER CONVERTING
50
├──10
│   ├──3
│   │   ├──2
│   │   │   └──0
│   │   └──5
│   │       └──6
│   └──25
│       ├──20
│       └──30
└──120
    ├──100
    │   ├──60
    │   └──110
    └──150
        ├──130
        └──250
```



```
BEFORE CONVERTING
100
└──200
    ├──150
    │   └──125
    │       └──110
    │           └──105
    └──300
        └──400

AFTER CONVERTING
150
├──110
│   ├──100
│   │   └──105
│   └──125
└──300
    ├──200
    └──400
```

TIME COMPLEXITY ANALYSIS

Q1-METHODS



```java
@SuppressWarnings("unchecked")
public BinarySearchTree<E> CombineArrayAndBT(BinaryTree<E> BT, E[] array){
    Stack<E> stack = new Stack<E>();
    BinarySearchTree<E> BST = new BinarySearchTree<E>();
    BinaryTree.Node<E> newRoot = cloneBinaryTree(BT.root);

    sortInstance.sort(array); //sort array

    for(int i=array.length-1;i>=0;i--) stack.push(array[i]); //push array to stack

    BST.root = newRoot;

    BST.inOrderTraversePassArray(BST.root, stack);

    return BST;
}
```

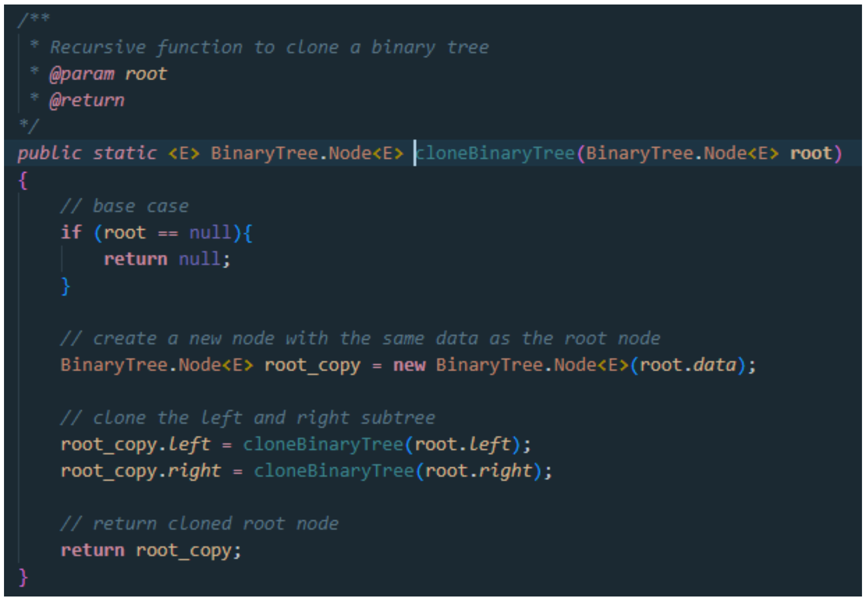
cloneBinaryTree() ==> | Best case : theta(logn) | worst case : theta(n) | general case : O(n)

sort(array) ==> | Best case : theta(n*logn) | worst case : theta(n^2) | general case : O(n) | average case : O(n*logn) |

for loop ==> theta(n)

inorderTraversePassArray( ) ==> O(n)

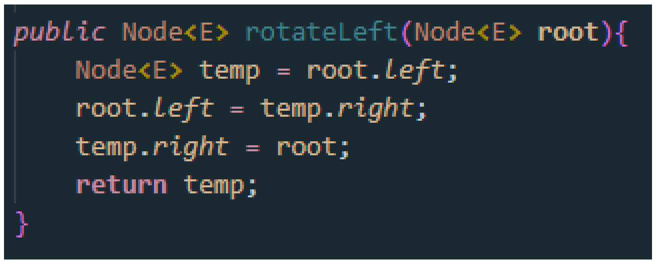
CombineArrayBT(…) ==> O(n^2)



```java
/**
 * Recursive function to clone a binary tree
 * @param root
 * @return
 */
public static <E> BinaryTree.Node<E> cloneBinaryTree(BinaryTree.Node<E> root)
{
    // base case
    if (root == null){
        return null;
    }

    // create a new node with the same data as the root node
    BinaryTree.Node<E> root_copy = new BinaryTree.Node<E>(root.data);

    // clone the left and right subtree
    root_copy.left = cloneBinaryTree(root.left);
    root_copy.right = cloneBinaryTree(root.right);

    // return cloned root node
    return root_copy;
}
```

cloneBinaryTree() == | Best case : theta(logn) | worst case : theta(n) | general case : O(n) |

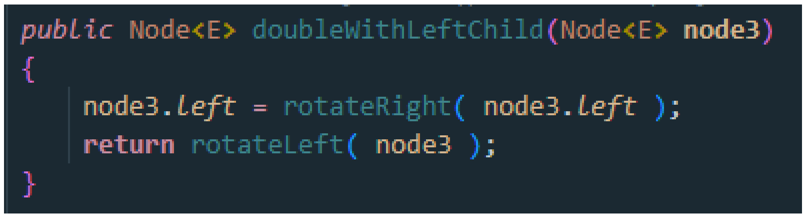
Q2-METHODS



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

rotateLeft(…) ==> theta(1)

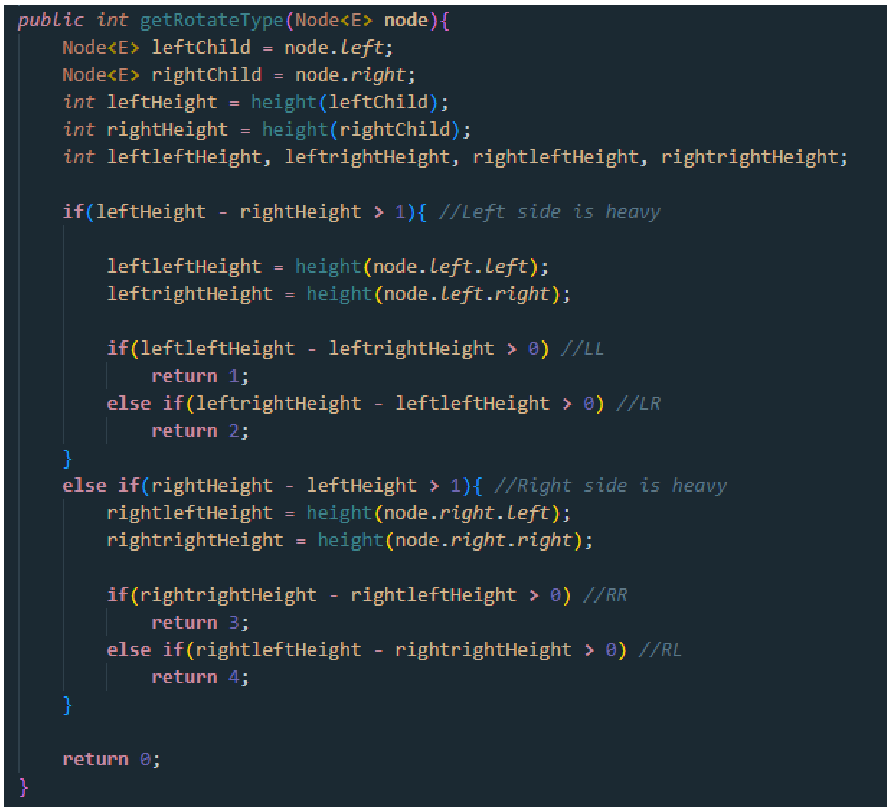
rotateRight is same.



```java
public Node<E> doubleWithLeftChild(Node<E> node3)
{
    node3.left = rotateRight( node3.left );
    return rotateLeft( node3 );
}
```

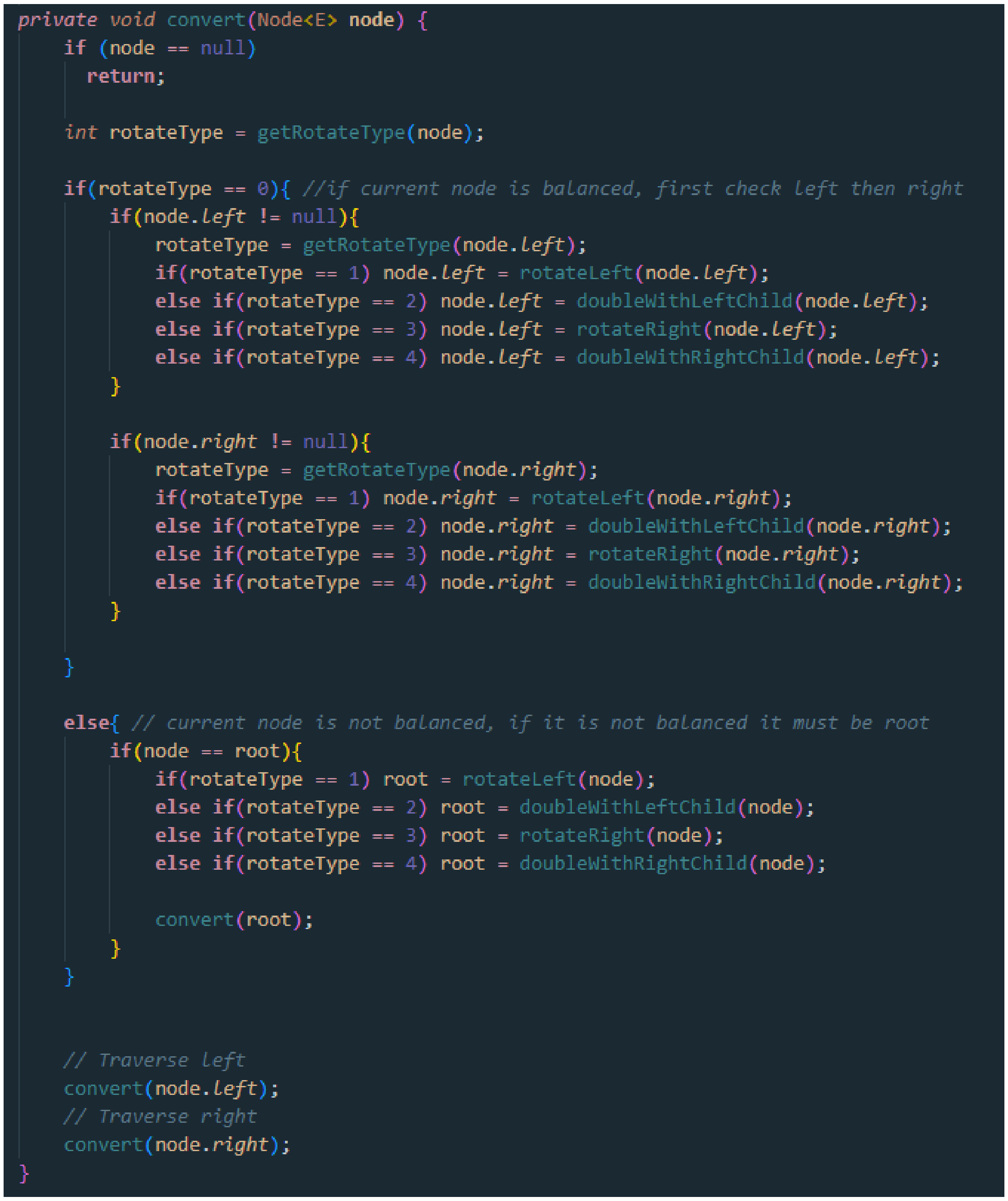
doubleWithLeftChild(…) ==> theta(1)

doubleWithRightChild is same.



```java
public int getRotateType(Node<E> node){
    Node<E> leftChild = node.left;
    Node<E> rightChild = node.right;
    int leftHeight = height(leftChild);
    int rightHeight = height(rightChild);
    int leftleftHeight, leftrightHeight, rightleftHeight, rightrightHeight;

    if(leftHeight - rightHeight > 1){ //Left side is heavy

        leftleftHeight = height(node.left.left);
        leftrightHeight = height(node.left.right);

        if(leftleftHeight - leftrightHeight > 0) //LL
            return 1;
        else if(leftrightHeight - leftleftHeight > 0) //LR
            return 2;
    }
    else if(rightHeight - leftHeight > 1){ //Right side is heavy
        rightleftHeight = height(node.right.left);
        rightrightHeight = height(node.right.right);

        if(rightrightHeight - rightleftHeight > 0) //RR
            return 3;
        else if(rightleftHeight - rightrightHeight > 0) //RL
            return 4;
    }

    return 0;
}
```

Calling height method
Height(…) ==> O(n)
getRotateType(…) ==> O(n)



```java
private void convert(Node<E> node) {
    if (node == null)
      return;

    int rotateType = getRotateType(node);

    if(rotateType == 0){ //if current node is balanced, first check left then right
        if(node.left != null){
            rotateType = getRotateType(node.left);
            if(rotateType == 1) node.left = rotateLeft(node.left);
            else if(rotateType == 2) node.left = doubleWithLeftChild(node.left);
            else if(rotateType == 3) node.left = rotateRight(node.left);
            else if(rotateType == 4) node.left = doubleWithRightChild(node.left);
        }

        if(node.right != null){
            rotateType = getRotateType(node.right);
            if(rotateType == 1) node.right = rotateLeft(node.right);
            else if(rotateType == 2) node.right = doubleWithLeftChild(node.right);
            else if(rotateType == 3) node.right = rotateRight(node.right);
            else if(rotateType == 4) node.right = doubleWithRightChild(node.right);
        }

    }

    else{ // current node is not balanced, if it is not balanced it must be root
        if(node == root){
            if(rotateType == 1) root = rotateLeft(node);
            else if(rotateType == 2) root = doubleWithLeftChild(node);
            else if(rotateType == 3) root = rotateRight(node);
            else if(rotateType == 4) root = doubleWithRightChild(node);

            convert(root);
        }
    }


    // Traverse left
    convert(node.left);
    // Traverse right
    convert(node.right);
}
```

getRotateType(…) ==> O(n)

rotateLeft(…) ==> theta(1)

doubleWithLeftChild(…) ==> theta(1)

doubleWithRightChild(…) ==> theta(1)

convert(…) ==> O(n^2)