

**Gebze Technical University
Computer Engineering**

CSE 222 - 2018 Spring

HOMEWORK 6 REPORT

**FERHAT ŞİRİN
161044080**

Course Assistant: Fatma Nur Esirci

1 Worst RedBlack Tree

1.1 Problem Solution Approach

```
public class RedBlackTree<E extends Comparable<E>> extends  
    BinarySearchTreeWithRotate<E>{
```

RedBlackTree bir binary tree ve aynı zamanda bir search tree olduğu için BinarySearchTreeWithRotate sınıfını extend eder ve bu sınıfı kullanmak için comparable edilebilen bir türde nesne ile kullanmak zorunludur. RedBlackTree'de ağacın dengeli halde olması için ekleme sırasında gerekli rotate işlemleri BinarySearchTreeWithRotate sınıfından gelen methodlar yardımıyla yapılır.

```
public boolean add(E item) {  
    if (root == null) {  
        root = new RedBlackNode<E>(item);  
        ((RedBlackNode<E>) root).isRed = false; // root is black.  
        return true;  
    } else {  
        root = add((RedBlackNode<E>) root, item);  
        ((RedBlackNode<E>) root).isRed = false; // root is always black.  
        return addReturn;  
    }  
}
```

Public add methodu ile dışarıdan eklemek istenilen değer eklenilir. RedBlack Tree yapısı binary search tree olduğu için recursive halde ekleme yapmak daha kolaydır. Bu yüzden önce root a bakılır eğer root boş ise ilk ekleme yapılır fakat root dolu ise rootun dallarından birine ekleme yapılması gerektiği için recursive add methodu çağrılır.

```
private Node<E> add(RedBlackNode<E> localRoot, E item) {  
    if (item.compareTo(localRoot.data) == 0) {  
        // item already in the tree.  
        addReturn = false;  
        return localRoot;  
    } else if (item.compareTo(localRoot.data) < 0) {  
        // item < localRoot.data.  
        if (localRoot.left == null) {  
            // Create new left child.  
            localRoot.left = new RedBlackNode<E>(item);  
            addReturn = true;  
            return localRoot;  
        } else { // Need to search.  
            // Check for two red children, swap colors if found.  
            moveBlackDown(localRoot);  
            // Recursively add on the left.  
            localRoot.left = add((RedBlackNode<E>) localRoot.left, item);  
            // See whether the left child is now red  
            if (((RedBlackNode<E>) localRoot.left).isRed) {  
                if (localRoot.left.left != null && ((RedBlackNode<E>) localRoot.left.left).isRed) {
```

```

        // Left-left grandchild is also red.
        // Single rotation is necessary.
        ((RedBlackNode<E>) localRoot.left).isRed = false;
        localRoot.isRed = true;
        return rotateRight(localRoot);
    } else if (localRoot.left.right != null && ((RedBlackNode<E>)
localRoot.left.right).isRed) {
        // Left-right grandchild is also red.
        // Double rotation is necessary.
        localRoot.left = rotateLeft(localRoot.left);
        ((RedBlackNode<E>) localRoot.left).isRed = false;
        localRoot.isRed = true;
        return rotateRight(localRoot);
    }
}
return localRoot;
}
} else {
    if (localRoot.right == null) {
        // Create new left child.
        localRoot.right = new RedBlackNode<E>(item);
        addReturn = true;
        return localRoot;
    } else { // Need to search.
        // Check for two red children, swap colors if found.
        moveBlackDown(localRoot);
        // Recursively add on the left.
        localRoot.right = add((RedBlackNode<E>) localRoot.right, item);
        // See whether the left child is now red
        if (((RedBlackNode<E>) localRoot.right).isRed) {
            if (localRoot.right.right != null && ((RedBlackNode<E>)
localRoot.right.right).isRed) {
                // Left-left grandchild is also red.
                // Single rotation is necessary.
                ((RedBlackNode<E>) localRoot.right).isRed = false;
                localRoot.isRed = true;
                return rotateLeft(localRoot);
            } else if (localRoot.right.left != null && ((RedBlackNode<E>)
localRoot.right.left).isRed) {
                // Left-right grandchild is also red.
                // Double rotation is necessary.
                localRoot.right = rotateLeft(localRoot.right);
                ((RedBlackNode<E>) localRoot.right).isRed = false;
                localRoot.isRed = true;
                return rotateRight(localRoot);
            }
        }
    }
    return localRoot;
}
}

```

Recursive add methodu ilk olarak normal bir binary search tree gibi elemanın karşılaştırdığı büyük ise sağa küçük ise sola eklenmesini sağlar. Eklenilecek yer belirlendikten sonra o bölge boş ise eleman red olarak işaretlenip eklenir. Eğer dolu ise recursive method yeniden çağrılır ve uygun alana gelinceye kadar bu devam eder. Bu süreç içinde eğer iki child nodu red olarak işaretlenmiş tree varsa 2 child node black, root node ise red olarak işaretlenir bu sayede red nodun sadece siyah

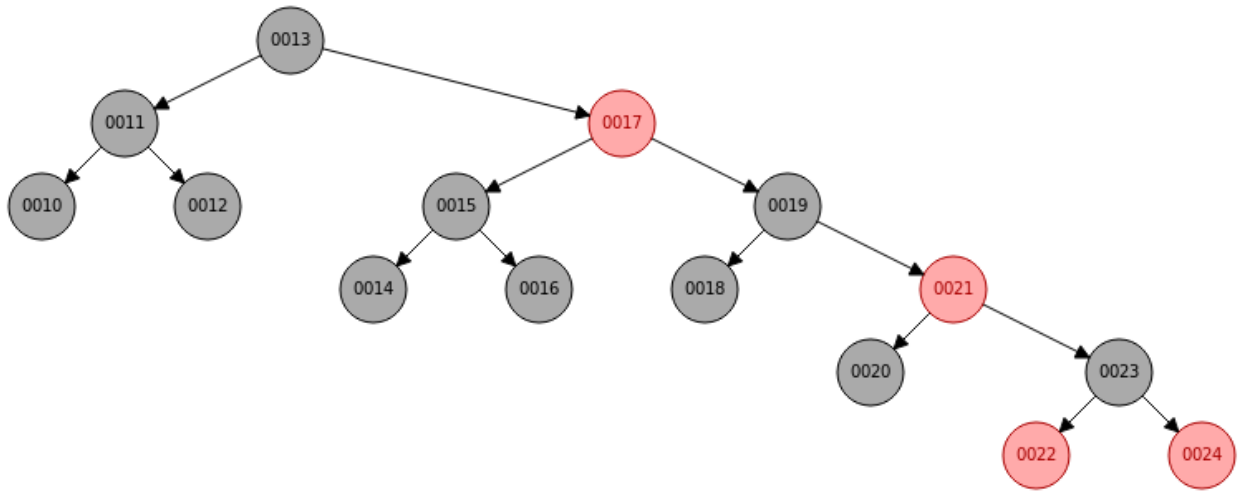
child nodları olması sağlanır ve RedBlack Tree kuralı bozulmamış olur. Eklenilecek alan bulunduğunda ise eleman ilk olarak red olarak işaretlenip eklenir. Ekleme bittikten sonra tekrar geriye dönüştürme arka arkaya red node olup olmadığına bakılır eğer var ise rotate yapılarak RedBlack Tree ağaç özelliği korunur.

1.2 Test Cases

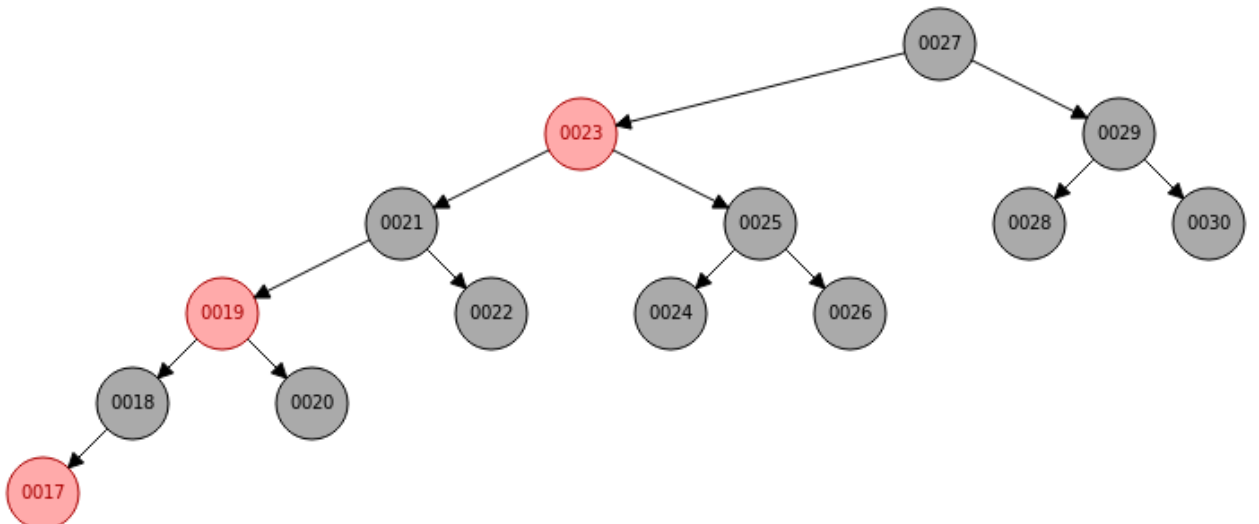
En kötü durum testi için eklenicek olan elemanlar sıralı olarak verilmiştir. Bu sayede olabilecek tüm eklemelerde rotate yaptırılmıştır.

Test durumu RedBlackTreeTest classında main metodundadır.

Test 1 Tree:

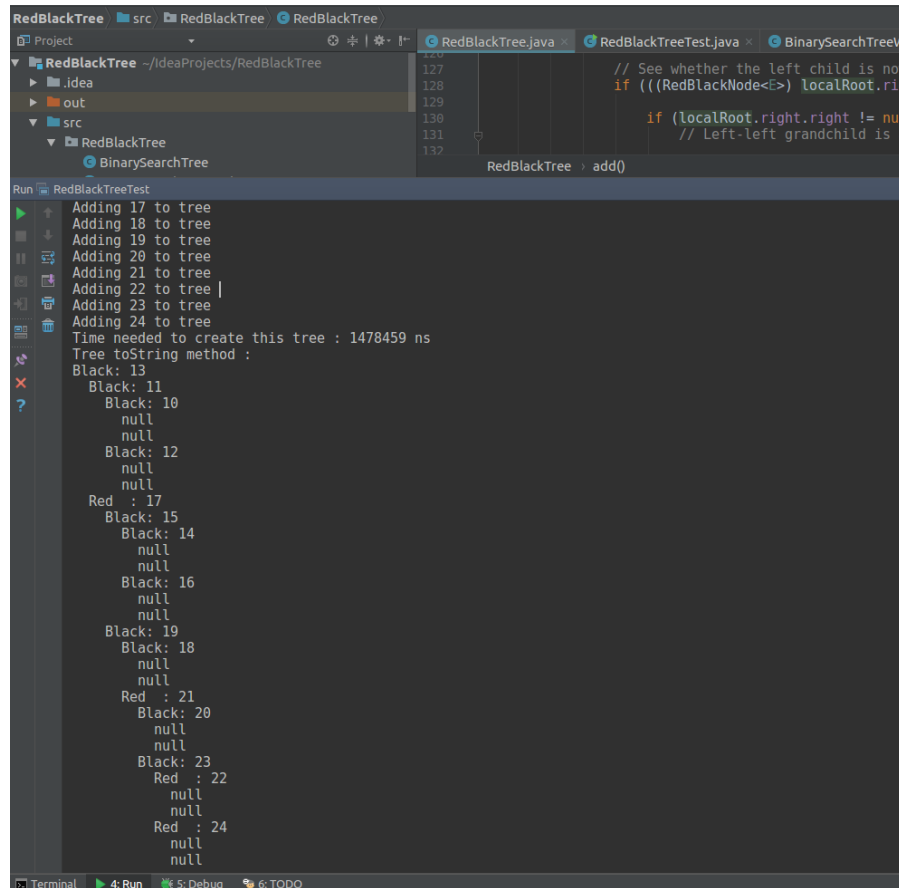


Test 2 Tree :



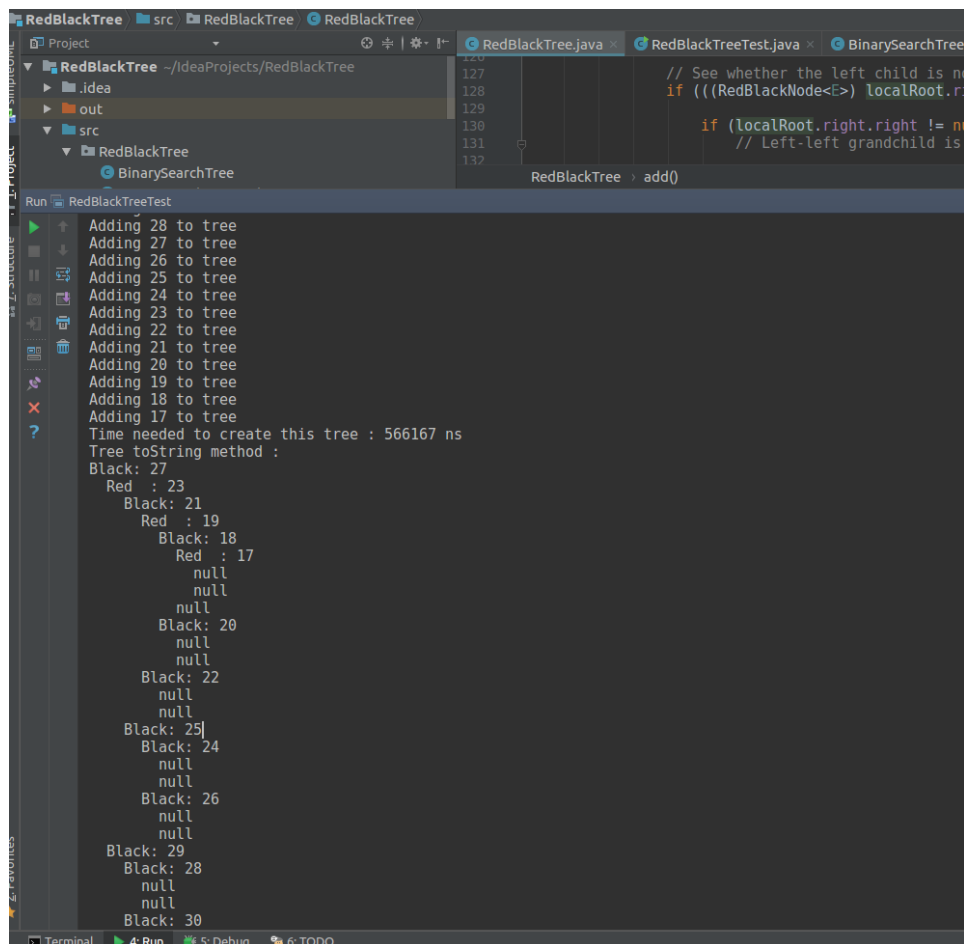
1.3 Running Commands and Results

Test 1:



```
Run RedBlackTreeTest
Adding 17 to tree
Adding 18 to tree
Adding 19 to tree
Adding 20 to tree
Adding 21 to tree
Adding 22 to tree
Adding 23 to tree
Adding 24 to tree
Time needed to create this tree : 1478459 ns
Tree toString method :
Black: 13
  Black: 11
    Black: 10
      null
      null
    Black: 12
      null
      null
  Red : 17
    Black: 15
      Black: 14
        null
        null
      Black: 16
        null
        null
    Black: 19
      Black: 18
        null
        null
      Red : 21
        Black: 20
          null
          null
        Black: 23
          Red : 22
            null
            null
          Red : 24
            null
            null
```

Test 2 :



```
Run RedBlackTreeTest
Adding 28 to tree
Adding 27 to tree
Adding 26 to tree
Adding 25 to tree
Adding 24 to tree
Adding 23 to tree
Adding 22 to tree
Adding 21 to tree
Adding 20 to tree
Adding 19 to tree
Adding 18 to tree
Adding 17 to tree
Time needed to create this tree : 566167 ns
Tree toString method :
Black: 27
  Red : 23
    Black: 21
      Red : 19
        Black: 18
          Red : 17
            null
            null
          Black: 20
            null
            null
        Black: 22
          null
          null
        Black: 25
          Black: 24
            null
            null
          Black: 26
            null
            null
        Black: 29
          Black: 28
            null
            null
          Black: 30
```

2 BinarySearch method

2.1 Problem Solution Approach

```
public class BTree<E extends Comparable<E>> implements SearchTree<E>
{
```

B tree bir search tree olduğu için SearchTree sınıfını implements eder ve bu interface de tanımlı olan add find remove gibi methodları implement eder. B treeyi comparable edilebilen türde bir nesne ile kullanmak zorunludur. B treede amaç büyük bir veriyi diskten tek seferde çekip kullanmaktır. Bu yüzden B tree nodeleri birden fazla değer barındırır. B treede oluşturulurken order değeri belirtilir ve her bir node en fazla order değeri kadar child node sahip olabilir ve her bir node order ın bir eksiği kadar elemana sahip olabilir. Bu düzenin sağlanması için B treede split işlemleri yapılır eğer bir node dolduğunda node parçalara ayrılır.

```
public boolean add(E item) {
    if (order == 0) {
        throw new IllegalStateException("Must set order first");
    }
    if (root == null) {
        root = new Node<E>(order);
        root.data[0] = item;
        root.size = 1;
        System.out.println("Adding "+item+" to tree");
        return true;
    }
    newChild = null;
    boolean result = insert(root, item);
    if (newChild != null) {
        Node<E> newRoot = new Node<E>(order);
        newRoot.child[0] = root;
        newRoot.child[1] = newChild;
        newRoot.data[0] = newParent;
        newRoot.size = 1;
        root = newRoot;
    }
    return result;
}
```

Add methodu ile B tree de ekleme yapılır. İlk olarak root node bakılır boş ise eklenir. Her node order -1 kadar eleman tutabilir. Eğer root dolu ise insert methodu çağrılır ve ekleme insert metodunda yapılır.

```
private boolean insert(Node<E> root, E item) {
    int index = binarySearch(item, root.data, 0, root.size-1);
    if (index != root.size && item.compareTo(root.data[index]) == 0) {
        return false;
    }
    if (root.child[index] == null) {
        System.out.println("Adding "+item+" to tree");
        if (root.size < order - 1) {
            insertIntoNode(root, index, item, null);
        }
    }
}
```

```

        newChild = null;
    } else {
        splitNode(root, index, item, null);
    }
    return true;
} else {
    boolean result = insert(root.child[index], item);
    if (newChild != null) {
        if (root.size < order - 1) {
            insertIntoNode(root, index, newParent, newChild);
            newChild = null;
        } else {
            splitNode(root, index, newParent, newChild);
        }
    }
    return result;
}
}

```

Recursive insert methodu ilk olarak binarySearch methodu çağrılıp bu elemanın eklemek için uygun olan yerin indexi istenir. B tree de nodelerin elemanları sıralı olduğu için eklenilecek eleman doğru sırada olmalıdır. Eğer döndürülen index te aynı eleman varsa tekrar ekleme yapılmaz false döndürülür fakat yer boş ise ekleme yapılır. Eğer yer dolu fakat eleman farklı ise bu durumda child nodelere geçilir ve aynı işlem recursive olarak yapılır. Eklemeden önce node tamamı dolu ise node split yapılarak parçalanır ve ekleme öyle yapılır.

```

private int binarySearch(E item,E[] data,int start,int end){
    if(start > end) {
        return start;
    }
    int middle =(start+end)/2;
    if(data[middle] ==null){
        return middle;
    }
    int comp =item.compareTo(data[middle]);
    if(comp < 0){
        return binarySearch(item,data,start,middle-1);
    }
    else if(comp > 0){
        return binarySearch(item,data,middle+1,end);
    }
    else {
        return middle;
    }
}

```

Binary search tree ile elemanın nereye eklenmesi gerektiğine karar verilir. Data arrayı bir nodun içindeki veridir. İlk olarak ortanca eleman ile eklenicek eleman karşılaştırılır eğer küçükse sol tarafa büyükse sağ tarafa recursive olarak gidilir. Eğer ortance değer boş çıkarsa o index döndürülür eğer başlangıç değeri arrayın bitiş değerini geçerse o durumda başlangıç değeri döndürülür bu durum node un child nodelerına geçilmesi gerektiği anlamına gelir. Eğer eleman varsa elemanın olduğu index döndürülür fakat aynı eleman tekrar eklenmez.

```

private void splitNode(Node<E> node, int index, E item, Node<E> child)
{
    // Create new child
    newChild = new Node<E>(order);
    // Determine number of items to move
    int numToMove = (order - 1) - ((order - 1) / 2);
    // If insertion point is to the right half, move one less item
    if (index > (order - 1) / 2) {
        numToMove--;
    }
    // Move items and their children
    System.arraycopy(node.data, order - numToMove - 1,
        newChild.data, 0, numToMove);
    System.arraycopy(node.child, order - numToMove,
        newChild.child, 1, numToMove);
    node.size = order - numToMove - 1;
    newChild.size = numToMove;
    // Insert new item
    if (index == ((order - 1) / 2)) { // Insert into middle
        newParent = item;
        newChild.child[0] = child;
    } else {
        if (index < ((order - 1) / 2)) { // Insert into the left
            insertIntoNode(node, index, item, child);
        } else {
            insertIntoNode(newChild, index - ((order - 1) / 2) - 1,
                item, child);
        }
        // The rightmost item of the node is the new parent
        newParent = node.data[node.size - 1];
        // Its child is the left child of the split-off node
        newChild.child[0] = node.child[node.size];
        node.size--;
    }
    // Remove contents and references to moved items
    for (int i = node.size; i < node.data.length; i++) {
        node.data[i] = null;
        node.child[i + 1] = null;
    }
}

```

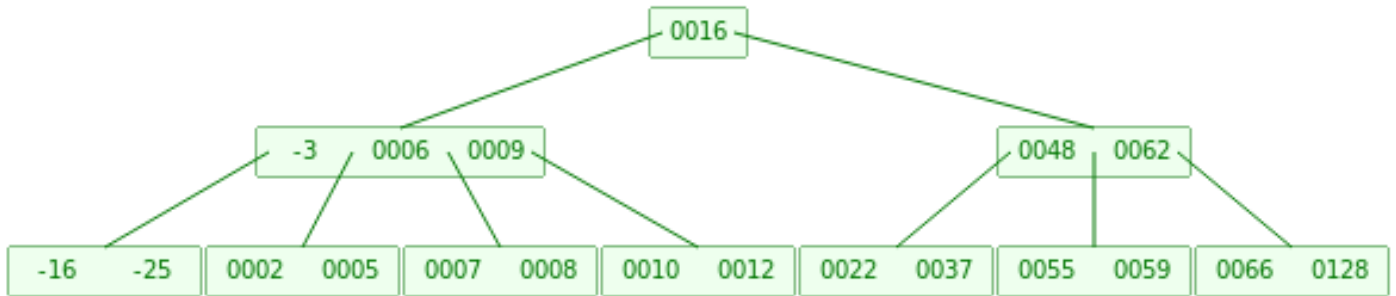
Eğer eklenmek istenen node dolu ise o node split yapılarak parçalara ayrılır. Her bir node en fazla order -1 kadar eleman tutabilir. Node yarısı yeni bir node aktarılır. Ortadaki eleman oluşan nodun yeni parent nodu olur. Eğer önceden bir parent node var ise orta eleman o parent node eklenir. Ekleme işlemi sıraları olarak yapılır bu sayede her bir node datası sıralı bir şekilde korunur.

2.2 Test Cases

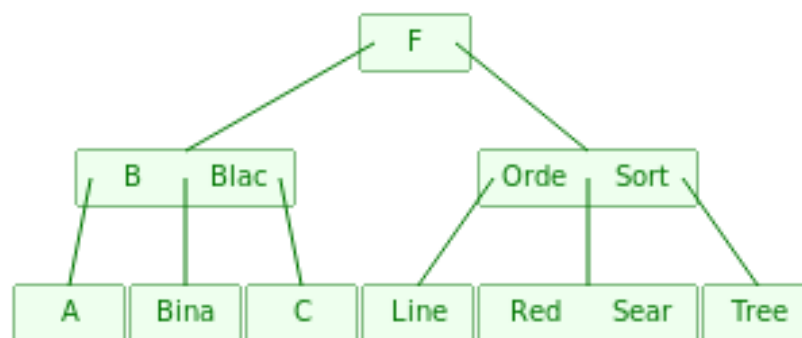
B tree de 2 ayrı test yazılmıştır. Test 1 : 5 order integer Test 2 : 3 order string değerleri ile yazılmıştır.

Test durumları BtreeTest sınıfının main methodundadır.

Test 1 sonuç :

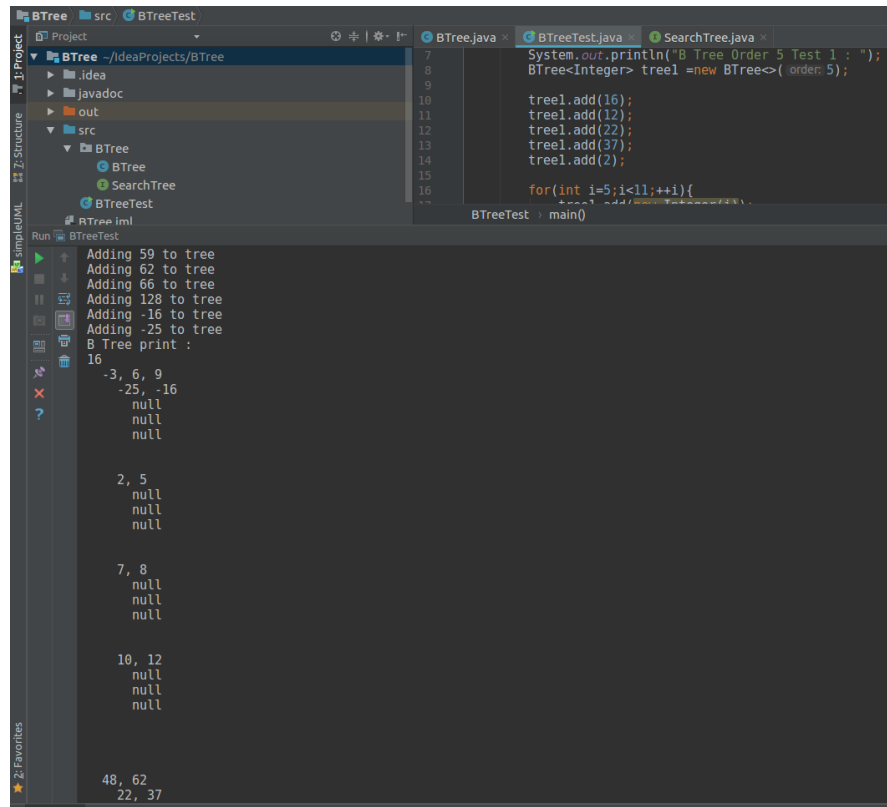


Test 2 sonuç :



2.3 Running Commands and Results

Test 1 :



```
System.out.println("B Tree Order 5 Test 1 : ");
BTree<Integer> tree1 = new BTree<>( order 5);

tree1.add(16);
tree1.add(12);
tree1.add(22);
tree1.add(37);
tree1.add(2);

for(int i=5;i<11;++i){
    tree1.add(i);
}
```

Run BTreeTest

```
Adding 59 to tree
Adding 62 to tree
Adding 66 to tree
Adding 128 to tree
Adding -16 to tree
Adding -25 to tree
B Tree print :
16
-3, 6, 9
-25, -16
null
null
null

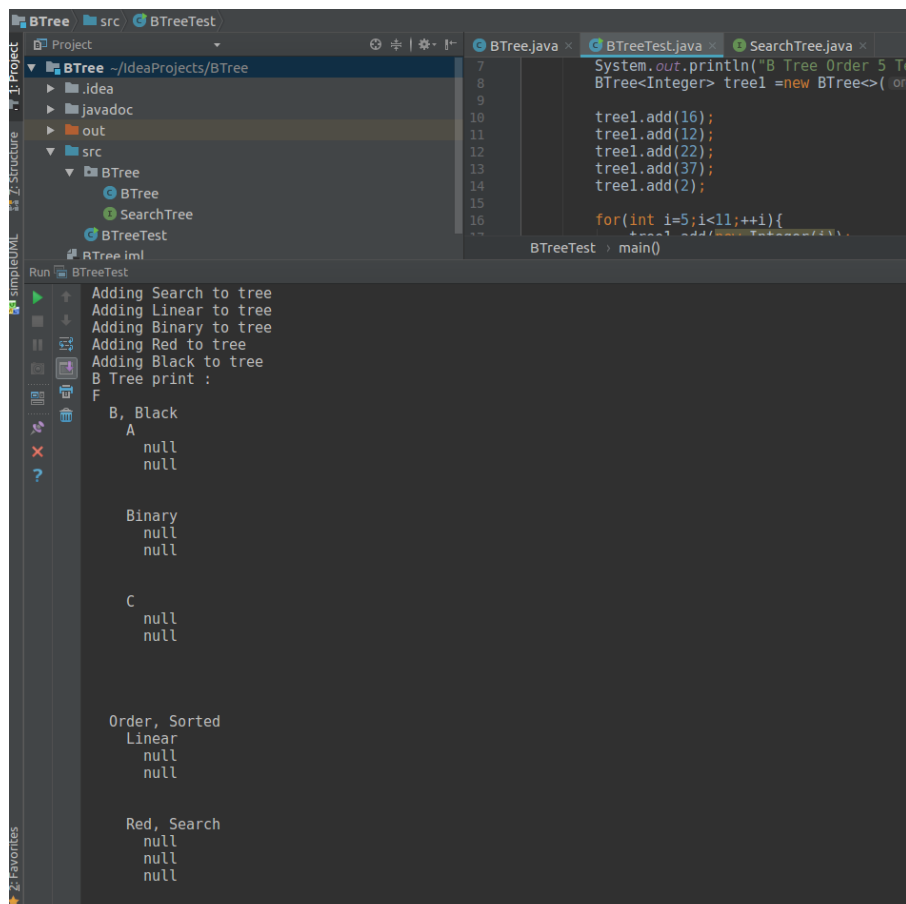
2, 5
null
null
null

7, 8
null
null
null

10, 12
null
null
null

48, 62
22, 37
```

Test 2 :



```
System.out.println("B Tree Order 5 Test 2 : ");
BTree<String> tree1 = new BTree<>( order 5);

tree1.add("16");
tree1.add("12");
tree1.add("22");
tree1.add("37");
tree1.add("2");

for(int i=5;i<11;++i){
    tree1.add(i);
}
```

Run BTreeTest

```
Adding Search to tree
Adding Linear to tree
Adding Binary to tree
Adding Red to tree
Adding Black to tree
B Tree print :
F
B, Black
A
null
null

Binary
null
null

C
null
null

Order, Sorted
Linear
null
null

Red, Search
null
null
null
```

3 Project 9.5 in book

3.1 Problem Solution Approach

```
public class AVLTree<E extends Comparable<E>>
    extends BinarySearchTreeWithRotate<E> {
```

AVL tree bir binary search tree olduğu için BinarySearchTreeWithRotate sınıfını extend eder ve bu sayede hem binary ve search özelliğini korurken aynı zamanda kendini dengeli halde tutabilir. AVL treeyi comparable edilebilen türde bir nesne ile kullanmak zorunludur. AVL tree'de amaç ekleme sırası ne olursa olsun eklenen elemanlar ağaçta dengeli bir şekilde eklenecek. Bu yüzden AVL tree nodeleri balance değeri barındırır. AVL tree'de balance sağ ağacın uzunluğundan sol ağacın uzunluğu çıkarılarak bulunur ve bu sayı 1 değerini geçmemelidir. Balance değeri 2 olduğunda rebalance methodu ile ağaç dengeli duruma getirilir.

```
private AVLNode<E> add(AVLNode<E> localRoot, E item) {
    if (localRoot == null) {
        addReturn = true;
        increase = true;
        return new AVLNode<E>(item);
    }
    if (item.compareTo(localRoot.data) == 0) {
        // Item is already in the tree.
        increase = false;
        addReturn = false;
        return localRoot;
    }
    else if (item.compareTo(localRoot.data) < 0) {
        // item < data
        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; // Rebalance not needed.
    }
    else { // item > data
        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; // Rebalance not needed.
    }
}
```

Add methodu recursive yapıdadır. Eklenmek istenen eleman ilk olarak normal bir binary search tree'de nasıl ekleniyorsa o şekilde uygun yer bulunarak eklenir. Eğer küçükse sol, büyükse sağ ağaca bakılarak uygun yer bulunur ve eğer sol tarafa eklendiyse üsteki nodun balance değeri düşürülür eğer sağ tarafa eklenmişse üsteki nodun balance değeri artırılır. Sol taraf eğer -2 ve aşağı bir değere ulaşırsa rebalance işlemi yapılır bu değer sağ taraf için 2 ve üstüdür. Ağacın bir bölgesinde bozulma varsa direk düzeltilir ve ağacın sürekli dengeli olması sağlanır.

```
private AVLNode<E> rebalanceLeft(AVLNode<E> localRoot) {
    System.out.println("Rebalancing left part of node "+localRoot.data);
    // 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.LEFT_HEAVY;
            leftRightChild.balance = AVLNode.BALANCED;
            localRoot.balance = AVLNode.BALANCED;
        } else if (leftRightChild.balance > AVLNode.BALANCED) {
            leftChild.balance = AVLNode.BALANCED;
            leftRightChild.balance = AVLNode.BALANCED;
            localRoot.balance = AVLNode.RIGHT_HEAVY;
        } else {
            leftChild.balance = AVLNode.BALANCED;
            localRoot.balance = AVLNode.BALANCED;
        }
        // Perform left rotation.
        localRoot.left = rotateLeft(leftChild);
    } else { //Left-Left case
        // In this case the leftChild (the new root)
        // and the root (new right child) will both be balanced
        // after the rotation.
        leftChild.balance = AVLNode.BALANCED;
        localRoot.balance = AVLNode.BALANCED;
    }
    // Now rotate the local root right.
    return (AVLNode<E>) rotateRight(localRoot);
}
```

RebalanceLeft ve RebalanceRight methodları ile ağacın sol ve sağ tarafının daha fazla gelmesi durumu düzeltilir. Eğer ağacın sol tarafında sağa göre 2 veya daha fazla eleman varsa ağaç sağa döndürülür ve balance değerinin sıfırlanması sağlanır. Ağaç sağa döndürülmeden önce ağacın sağ child nodunun balance değerine bakılır eğer o bölgede bir bozukluk var ise (Left-Right durumu) önce sağ ağaç sola döndürülür sonra parent node sağa döndürülür ve ağacın dengesi sağlanır.

```

private AVLNode<E> remove(AVLNode<E> localRoot, E item) {
    if (localRoot == null) {
        // item is not in the tree.
        deleteReturn = null;
        return localRoot;
    }
    // Search for item to delete.
    int compResult = item.compareTo(localRoot.data);
    if (compResult < 0) {
        // item is smaller than localRoot.data.
        localRoot.left = remove((AVLNode<E>)localRoot.left, item);
        if (decrease) {
            findBalance(localRoot);
            if (localRoot.balance != AVLNode.BALANCED) {
                decrease = false;
            }
            if (localRoot.balance > AVLNode.RIGHT_HEAVY) {
                localRoot = rebalanceRight(localRoot);
                findBalance(localRoot);
                return localRoot;
            }
        }
    }
    return localRoot;
} else if (compResult > 0) {
    // item is larger than localRoot.data.
    localRoot.right = remove((AVLNode<E>)localRoot.right, item);
    if (decrease) {
        findBalance(localRoot);
        if (localRoot.balance != AVLNode.BALANCED) {
            decrease = false;
        }
        if (localRoot.balance < AVLNode.LEFT_HEAVY) {
            localRoot = rebalanceLeft(localRoot);
            findBalance(localRoot);
            return localRoot;
        }
    }
}
return localRoot;
} else {
    // item is at local root.
    decrease = true;
    deleteReturn = localRoot.data;
    if (localRoot.left == null) {
        // If there is no left child, return right child
        // which can also be null.
        return (AVLNode<E>)localRoot.right;
    } else if (localRoot.right == null) {
        // If there is no right child, return left child.
        return (AVLNode<E>)localRoot.left;
    } else {
        // Node being deleted has 2 children, replace the data
        // with inorder predecessor.
        if (localRoot.left.right == null) {
            // The left child has no right child.

```

```

        // Replace the data with the data in the
        // left child.
        localRoot.data = localRoot.left.data;
        // Replace the left child with its left child.
        localRoot.left = localRoot.left.left;
    } else {
        // Search for the inorder predecessor (ip) and
        // replace deleted node's data with ip.
        localRoot.data = findLargestChild((AVLNode<E>)localRoot.left);
    }
    findBalance(localRoot);
    if(localRoot.left != null && ((AVLNode<E>)localRoot.left).balance
< AVLNode.LEFT_HEAVY)
        localRoot.left = rebalanceLeft((AVLNode<E>)localRoot.left);
    return localRoot;
}
}
}

```

Remove methodu binary search tree için yapılan methoda benzer bir yapıdadır. Çıkarılması istenen eleman önce recursive olarak aranır ve eğer var ise çıkarılır. Elemanın çıkarılması durumunda yerine gelmesi gereken eleman belirlenir. Eğer sağ nodu yok ise sol node direk gelir veya sol node yok ise sağ node gelir eğer her ikisi var ise sol nodun en büyük elemanı bulunarak çıkarılan yere koyulur. Çıkarma işlemi tamamlandıktan sonra en önemli kısım ağacın dengeli yapısı bozulmamalıdır. Bunun için çıkarılan node eğer dengeyi değiştirmiş ise parent nodun balance değeri bir arttırılır veya azaltılır. Bu değişiklik sonucu balance değerleri tekrar kontrol edilir eğer 2 değerine ulaşmışsa rebalance işlemi yapılarak ağacın dengesi korunur.

```

private void isAVL(AVLNode<E> node){
    if(node != null) {
        if(node.balance <=-2 || node.balance >=2){
            isAVL = false;
        }
        findBalance((AVLNode<E>)node.right);
        findBalance((AVLNode<E>)node.left);
    }
}

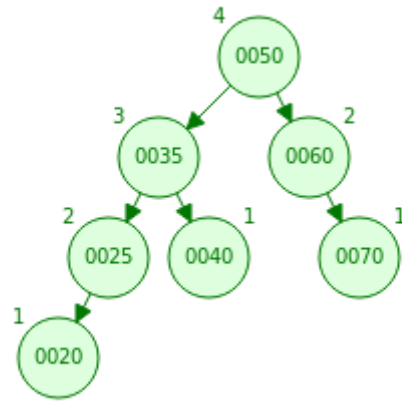
```

isAvl ile binary treeden aldığımız ağacın avl tree'ye uygun olup olmadığına bakarız. Eğer nodelerin balancı 2 veya 2 den büyükse bu ağaç avl ağaç değildir.

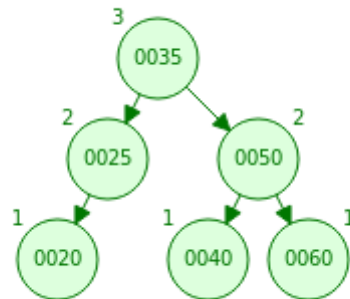
3.2 Test Cases

AVL tree için 2 ayrı test yazılmıştır. Test sırasında add remove ve bu methodlar dolayısıyla rebalance ve rotate işlemleri test edilmiş ve detaylı çıktı yazdırılmıştır. Test methodu AVLTreeTest sınıfının main methodundadır.

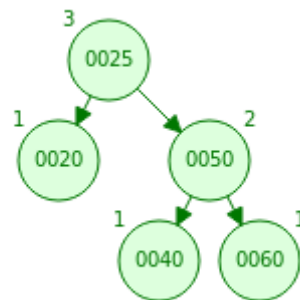
Test 1 :



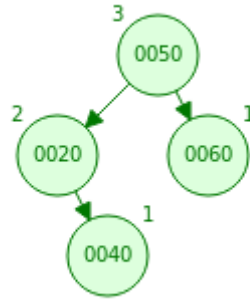
70 değeri çıkarıldıktan sonra :



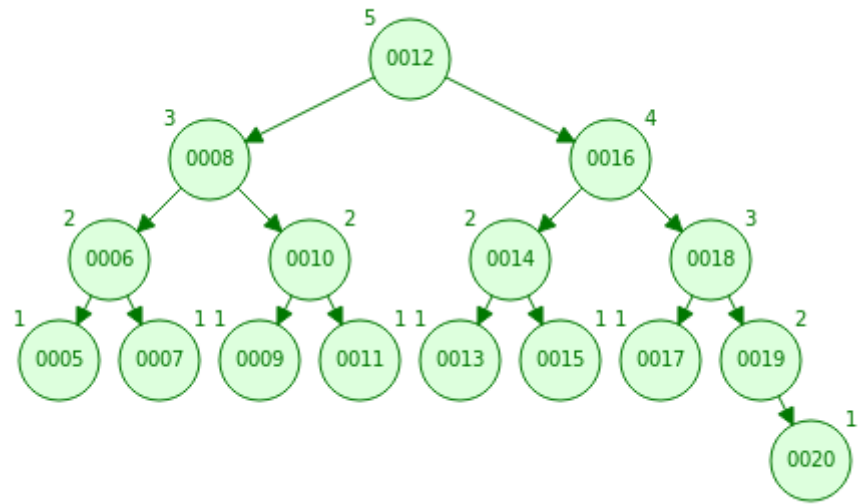
35 çıkarıldıktan sonra :



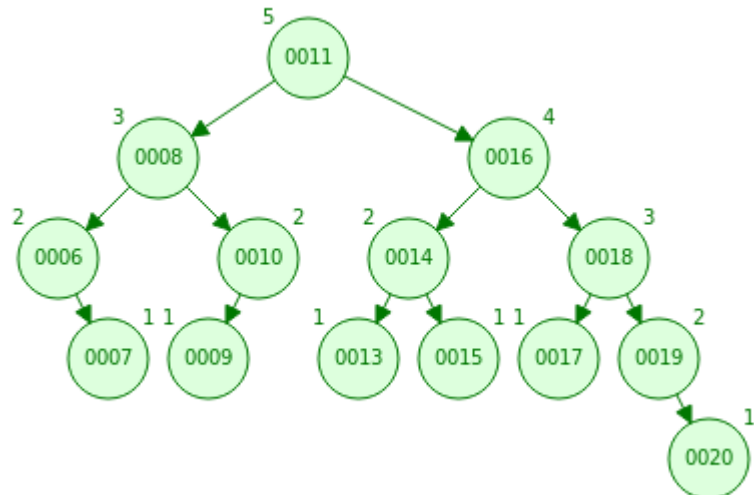
25 çıkarıldıktan sonra :



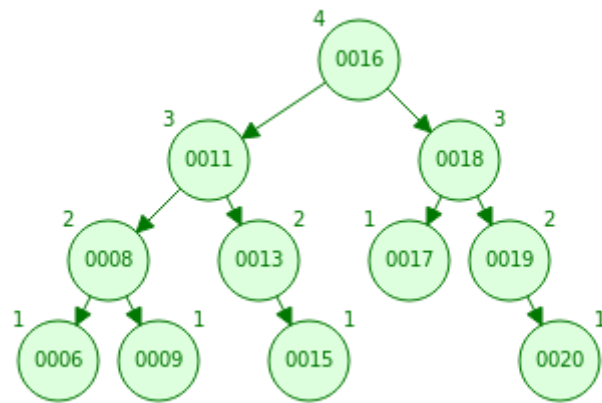
Test 2 :



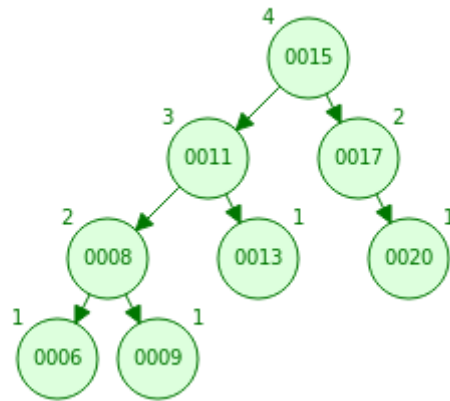
5 ve 12 çıkarıldıktan sonra :



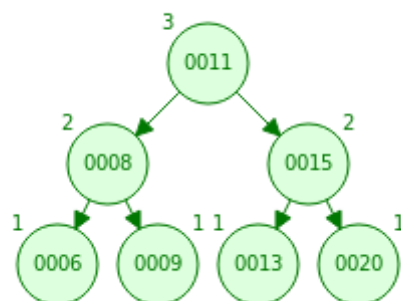
10, 14 ve 7 çıkarıldıktan sonra :



18, 19 ve 16 çıkarıldıktan sonra :

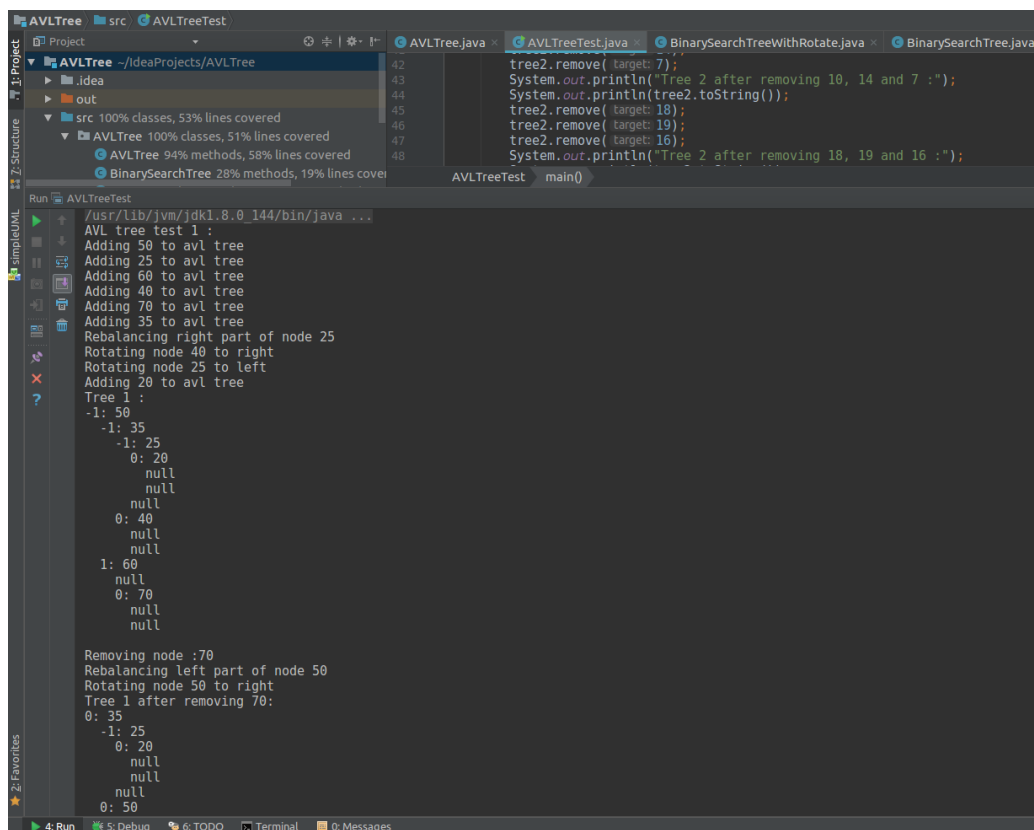


17 çıkarıldıktan sonra :



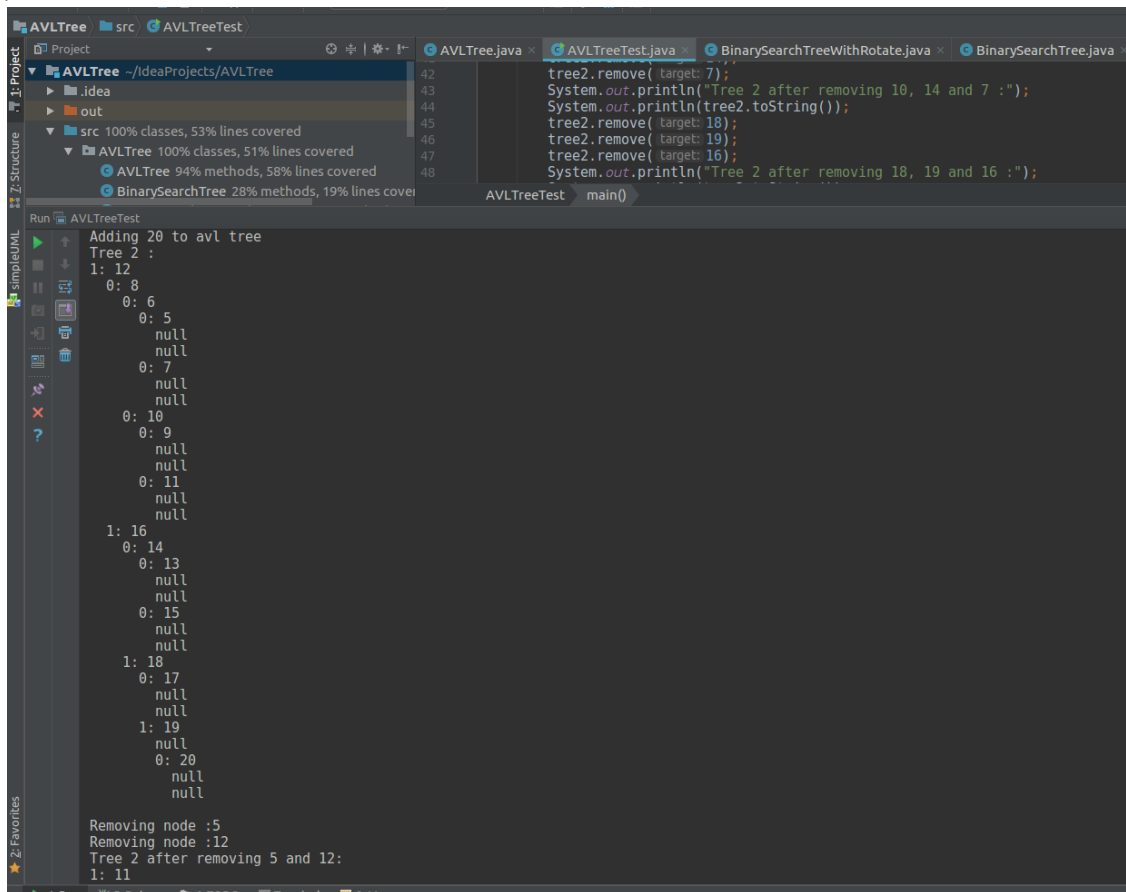
3.3 Running Commands and Results

Test 1 :



```
AVLTreeTest
Run
/usr/lib/jvm/jdk1.8.0_144/bin/java ...
AVL tree test 1 :
Adding 50 to avl tree
Adding 25 to avl tree
Adding 60 to avl tree
Adding 40 to avl tree
Adding 70 to avl tree
Adding 35 to avl tree
Rebalancing right part of node 25
Rotating node 40 to right
Rotating node 25 to left
Adding 20 to avl tree
Tree 1 :
-1: 50
  -1: 35
    -1: 25
      0: 20
        null
        null
      0: 40
        null
        null
    1: 60
      null
      0: 70
        null
        null
  Removing node :70
  Rebalancing left part of node 50
  Rotating node 50 to right
  Tree 1 after removing 70:
  0: 35
    -1: 25
      0: 20
        null
        null
    null
    0: 50
```

Test 2:



```
AVLTreeTest
Run
Adding 20 to avl tree
Tree 2 :
1: 12
  0: 8
    0: 6
      0: 5
        null
        null
      0: 7
        null
        null
    0: 10
      0: 9
        null
        null
      0: 11
        null
        null
  1: 16
    0: 14
      0: 13
        null
        null
      0: 15
        null
        null
    1: 18
      0: 17
        null
        null
      1: 19
        null
        0: 20
          null
          null
  Removing node :5
  Removing node :12
  Tree 2 after removing 5 and 12:
  1: 11
```