

**Gebze Technical University  
Computer Engineering**

**CSE 222 - 2018 Spring**

**HOMEWORK 4 REPORT**

**Sevgi BAYANSALDUZ  
151044076**

Course Assistant: Mehmet Burak KOCA

# 1 INTRODUCTION

## 1.1 Problem Definition

- **PART 1**

Genel ağaçların ikili ağaç temsilini oluşturan bir sınıfı, ikili ağaç (Binary Search) sınıfını genişleterek yapılmasıdır.

- **PART 2**

Çok boyutlu öğeler içeren genel arama ağacını oluşturan bir sınıfı, ikili arama ağacı (Binary search Tree) sınıfını genişleterek yapılmasıdır.

## 1.2 System Requirements

Proje intelij ide üzerinde yazılmıştır. Java 8 kullanılmıştır. Debian ve Windows 10 da çalıştırılabilir.

- **Part 1**

Genel ağaçların ikili ağaç temsili Part 1 `deki metodlar ile oluşturulurken aşağıdaki metodlar kullanılır. (Metodların alması gereken parametreler aşağıda verilmiştir.)

- `add(E item)`: Bu metod genel ağacın ilk elemanını eklemek için kullanılır, `genel(generic)` tipte parametre alır. (Part 1 i test etmek için ağaç oluşturulurken öncelikle bu metod kullanılmalı.)
- `add(E parentItem, E childItem)`: Bu metod `genel(generic)` tipte `parentItem` ve `childItem` alır.
- `levelOrderSearch(E target)`, `postOrderSearch(E target)`: Bu iki metod `genel(generic)` tipte parametre alır.
- `preOrderTraverse(Node node, StringBuilder sb, int depth)`: Bu metodun tipi `protected` tipindedir. Bu metodu test etmek için `toString()` metodunun çağırılması yeterlidir.

- **Part 2**

Çok boyutlu öğeler içeren genel arama ağacı oluşturulurken aşağıdaki metodlar kullanılır.

- `add(ArrayList<E> item)`: Bu metod parametre olarak koleksiyon(collection) alır. İlk eleman ağaca eklendikten sonra bu metod çağırılır.

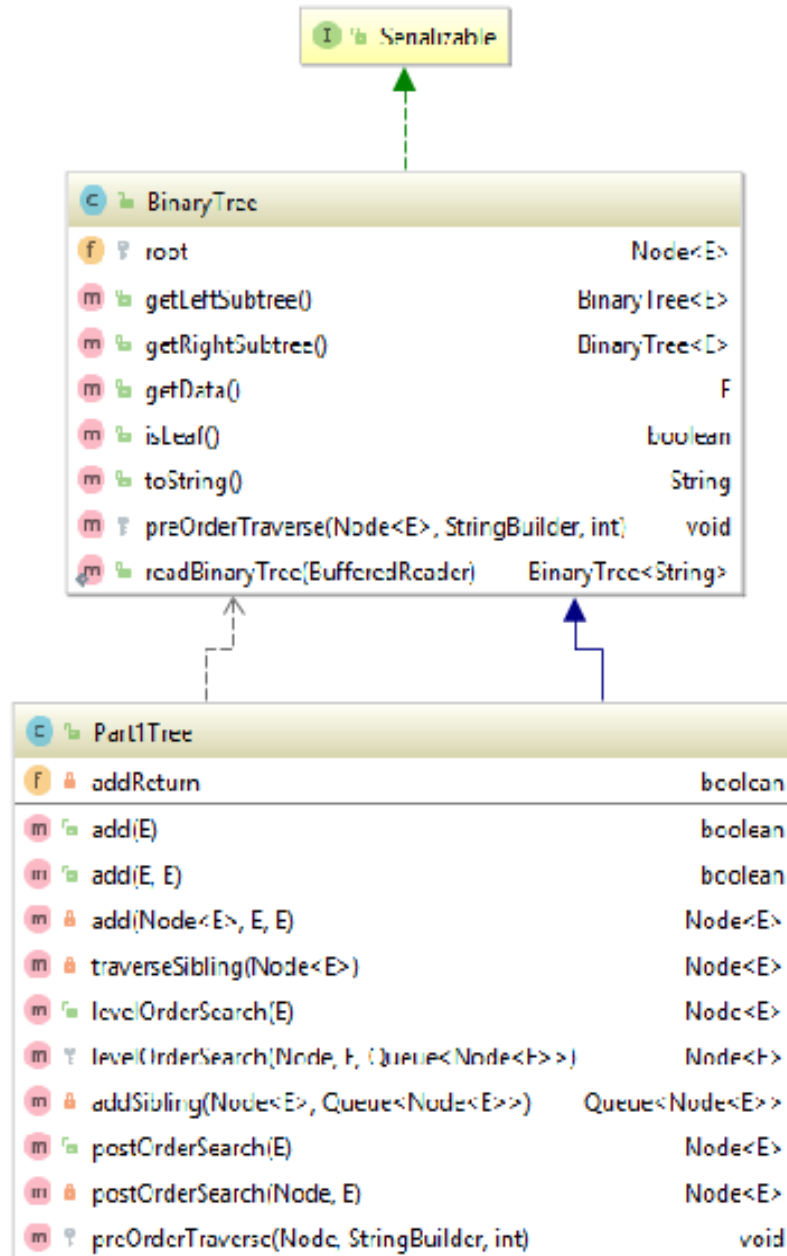
İlk elemanı sınıftaki yapıcı(constructor) metodları ile oluşturabilir. Örneğin:

```
node = new BINARYTree.Node<Integer> (item1);  
deneme = new MDSTree(node);
```

- `find(ArrayList<E> target)`, `contains (ArrayList<E> target)`, `delete (ArrayList<E> target)`, `remove (ArrayList<E> target)`: Bu dört metod parametre olarak koleksiyon(collection) alır.

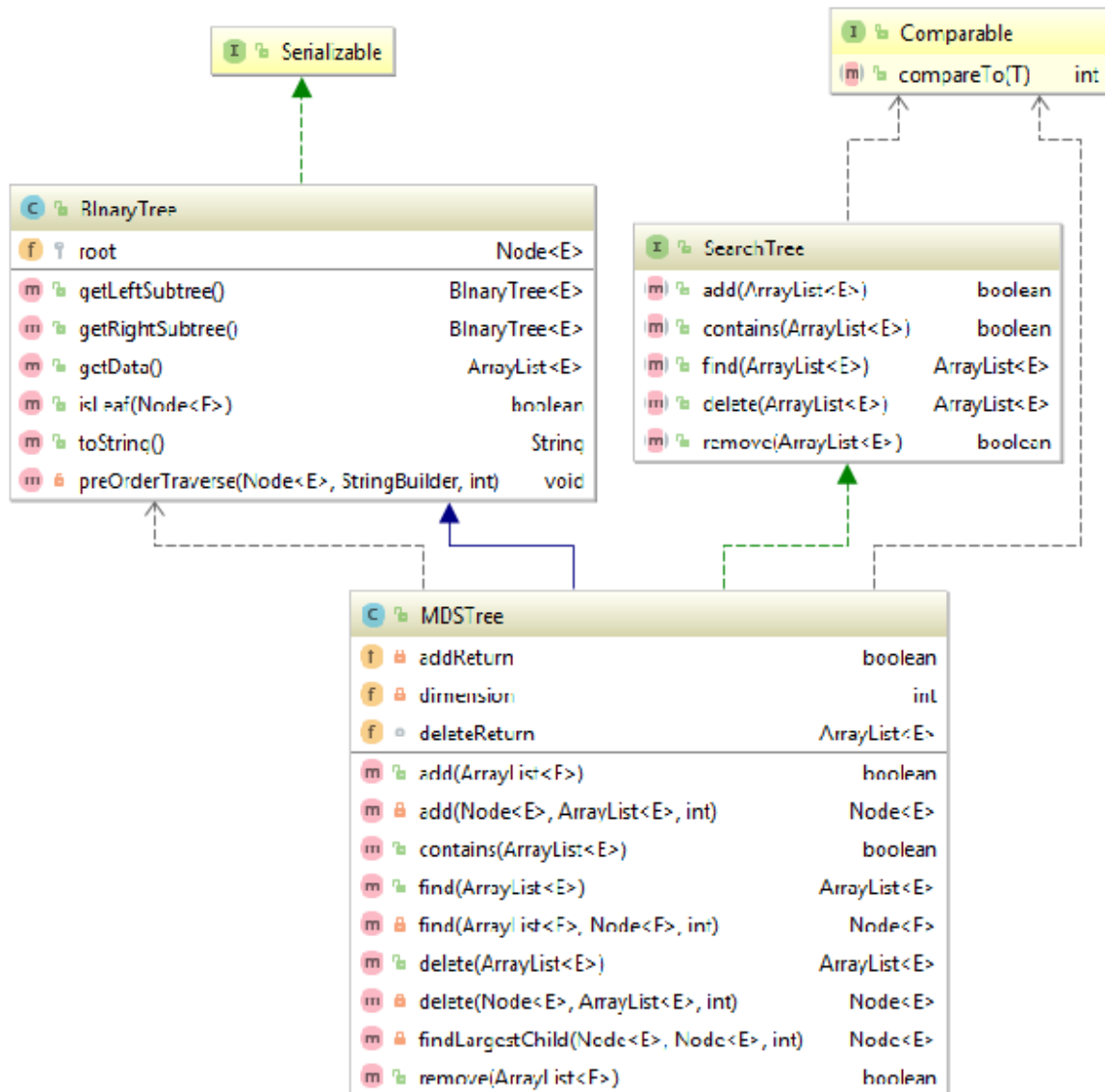
## 1.3 Class Diagrams

### ○ PART 1



Part1Tree sınıfı BinarySearch sınıfından genişler.

## ○ PART 2



MDSTree sınıfı BinaryTree sınıfından genişler(extend),SearchTree sınıfını uygular(implement).

### 1.4 Use Case Diagrams

Add use case diagrams if required.

### 1.5 Other Diagrams (optional)

Add other diagrams if required.

## 1.6 Problem Solution Approach

Part 1 için yapıyı oluştururken ikili ağaç (BinaryTree) yapısından yola çıktım. Metodlarımı yazarken ikili ağaç (BinaryTree) metodlarının algoritmalarından esinlenerek genel (general) ağaç metodları için yeni algoritmalar oluşturdum. Aynı işlevi Part2 için ikili arama ağacından esinlenerek (Binary Search Tree) gerçekleştirdim.

## 2 RESULT

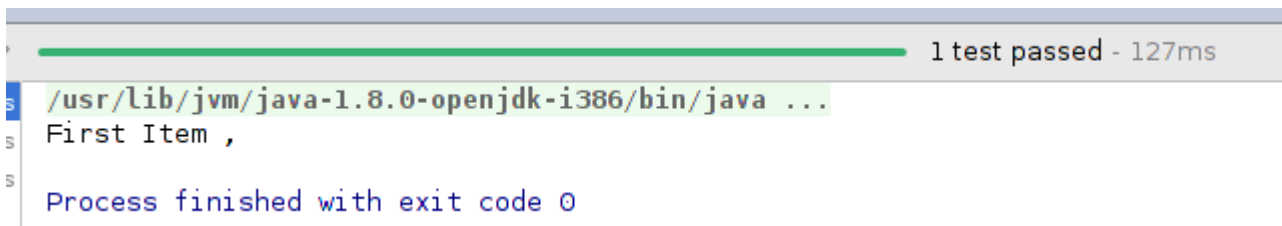
### 2.1 Test Cases

- **Part 1**
  - **add(E item)**

```
/**
 * Test for the add method that adds the first element into the tree.
 */
@Test
void add() {
    Part1Tree test=new Part1Tree<String>();
    test.add("First Item");
    System.out.println(test.toString());
}
```

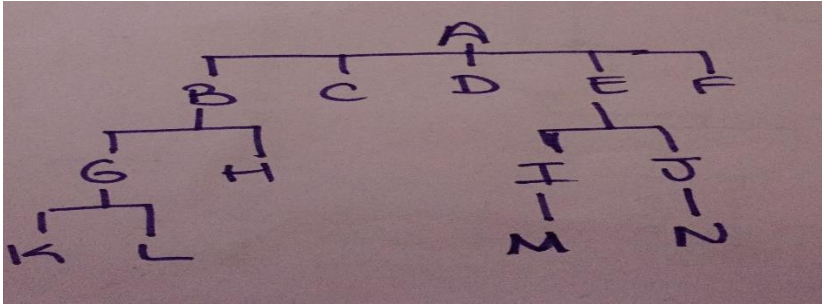
Bu test metodu ağaca ilk elemanı ekler. Ağacı ToString() metodu ile yazdırmak istediğimizde bu elemanı ekrana bastırmalıdır.

Bu test metodun sonucu:



```
/usr/lib/jvm/java-1.8.0-openjdk-i386/bin/java ...
First Item ,
Process finished with exit code 0
```

- add(E parentItem,E childItem) , levelOrderSearch(E target), postOrderSearch(E target), preOrderTraverse(Node node, StringBuilder sb, int depth) metodlarını test ederken aşağıdaki resimde gösterilen genel ağaç kullanıldı.



Resim 2.1.1

- `add(E parentItem,E childItem)` ve `preOrderTraverse(Node node, StringBuilder sb, int depth)`

```

/**
 * Test for the add method that adds the given ChildItem according to given ParentItem into the tree.
 */
@Test
void add1() {
    Part1Tree test=new Part1Tree<String>();
    test.add("A");
    test.add( parentItem: "A", childrenItem: "B");
    test.add( parentItem: "A", childrenItem: "C");
    test.add( parentItem: "A", childrenItem: "D");
    test.add( parentItem: "A", childrenItem: "E");
    test.add( parentItem: "A", childrenItem: "F");
    test.add( parentItem: "B", childrenItem: "G");
    test.add( parentItem: "B", childrenItem: "H");
    test.add( parentItem: "E", childrenItem: "I");
    test.add( parentItem: "E", childrenItem: "J");
    test.add( parentItem: "G", childrenItem: "K");
    test.add( parentItem: "G", childrenItem: "L");
    test.add( parentItem: "I", childrenItem: "M");
    test.add( parentItem: "J", childrenItem: "N");
    System.out.println("Test for add\n ToString method is called(ToString method uses PreOrderTraverse method):\n "
        +test.toString());
}

```

Bu test metodu verilen sıra ile elemanları ekler.Ağacı toString metodu ile ekran yazdırır.ToString metodu preORderTraverse metodunu kullandığında bu traverse metodu da test edilmiş olur.

Test metodunun sonucu:

```

1 test passed - 73ms

/usr/lib/jvm/java-1.8.0-openjdk-i386/bin/java ...
Test for add
ToString method is called(ToString method uses PreOrderTraverse method):
A , B , G , K , L , H , C , D , E , I , M , J , N , F ,

Process finished with exit code 0

```

- `levelOrderSearch(E target)`

```

@Test
void levelOrderSearch() {
    Part1Tree test=new Part1Tree<String>();
    test.add("A");
    test.add( parentItem: "A", childrenItem: "B");
    test.add( parentItem: "A", childrenItem: "C");
    test.add( parentItem: "A", childrenItem: "D");
    test.add( parentItem: "A", childrenItem: "E");
    test.add( parentItem: "A", childrenItem: "F");
    test.add( parentItem: "B", childrenItem: "G");
    test.add( parentItem: "B", childrenItem: "H");
    test.add( parentItem: "E", childrenItem: "I");
    test.add( parentItem: "E", childrenItem: "J");
    test.add( parentItem: "G", childrenItem: "K");
    test.add( parentItem: "G", childrenItem: "L");
    test.add( parentItem: "I", childrenItem: "M");
    test.add( parentItem: "J", childrenItem: "N");
    System.out.print("Level order traverse: ");
    System.out.print("\nReturn:"+test.levelOrderSearch( target: "N"));
}

```

Bu test metodu önce Resim 2.1.1 deki ağacı oluştur sonra bu ağaçta verilen elemanı level level arar.Parametre olarak verilen elemanı arar iken karşılaştırma yaptığı elemanları ekrana yazdırır.

Test metodunun sonucu:

```

1 test passed - 137ms
/usr/lib/jvm/java-1.8.0-openjdk-i386/bin/java ...
Level order traverse: A , B , C , D , E , F , G , H , I , J , K , L , M , N ,
Return:N
Process finished with exit code 0

```

- **postOrderSearch(E target)**

```

@Test
void postOrderSearch() {
    Part1Tree test=new Part1Tree<String>();
    test.add("A");
    test.add( parentItem: "A", childrenItem: "B");
    test.add( parentItem: "A", childrenItem: "C");
    test.add( parentItem: "A", childrenItem: "D");
    test.add( parentItem: "A", childrenItem: "E");
    test.add( parentItem: "A", childrenItem: "F");
    test.add( parentItem: "B", childrenItem: "G");
    test.add( parentItem: "B", childrenItem: "H");
    test.add( parentItem: "E", childrenItem: "I");
    test.add( parentItem: "E", childrenItem: "J");
    test.add( parentItem: "G", childrenItem: "K");
    test.add( parentItem: "G", childrenItem: "L");
    test.add( parentItem: "I", childrenItem: "M");
    test.add( parentItem: "J", childrenItem: "N");
    System.out.print("Post order traverse: ");
    System.out.print("\nReturn:"+test.postOrderSearch( target: "N"));
}

```

Bu test metodu önce Resim 2.1.1 deki ağacı oluştur sonra bu ağaçta verilen elemanı postOrder şeklinde arar.Parametre olarak verilen elemanı arar iken karşılaştırma yaptığı

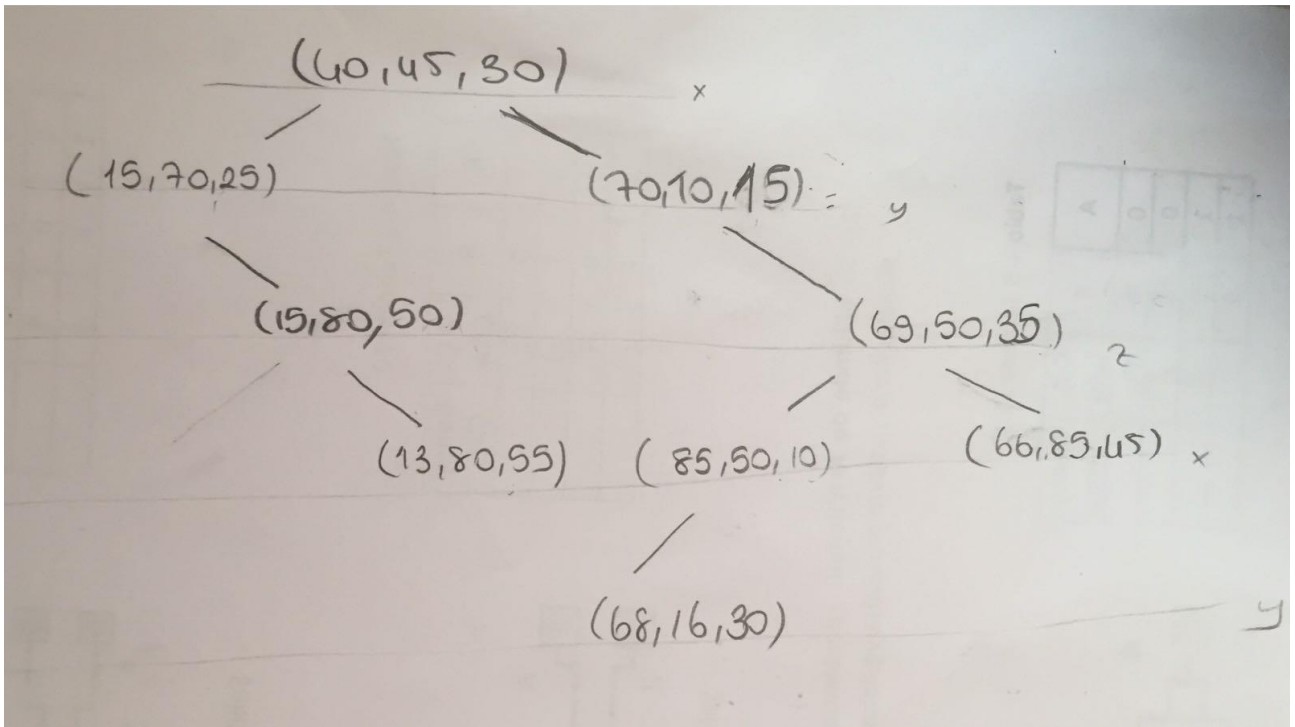
elemanları ekrana yazdırır.

Test metodunun sonucu:

```
1 test passed - 68ms
/usr/lib/jvm/java-1.8.0-openjdk-i386/bin/java ...
Post order traverse: K , L , G , H , B , C , D , M , I , N , J , E , F , A ,
Return:N
Process finished with exit code 0
```

- **Part 2**

Part 2`de bulunan metodların testleri için aşağıda görseli bulunan çok boyutlu ağaç kullanıldı.



**Resim 2.1.1**

- `add(ArrayList<E> item)`



```

ArrayList<Integer> item1 = new ArrayList<>(), item2=new ArrayList<>(), item3 =new ArrayList<>(), item4=new ArrayList<>(),
    item5=new ArrayList<>(), item6 =new ArrayList<>(), item7=new ArrayList<>(), item8=new ArrayList<>(),
    item9=new ArrayList<>(), item10=new ArrayList<>();
BinaryTree.Node<Integer> node;
MDSTree< Integer> deneme;
@Test
void add() {
    item1.add(40); item1.add(45); item1.add(30);
    item2.add(70); item2.add(10); item2.add(15);
    item3.add(69); item3.add(50); item3.add(35);
    item4.add(15); item4.add(70); item4.add(25);
    item5.add(85); item5.add(50); item5.add(10);
    item6.add(66); item6.add(85); item6.add(45);
    item7.add(15); item7.add(80); item7.add(50);
    item8.add(13); item8.add(80); item8.add(55);
    item9.add(69); item9.add(50); item9.add(35);
    item10.add(68); item10.add(16); item10.add(30);
    node = new BinaryTree.Node<Integer> (item1);
    deneme = new MDSTree(node);
    deneme.add(item2);
    deneme.add(item3); deneme.add(item4);
    deneme.add(item5); deneme.add(item6);
    deneme.add(item7); deneme.add(item8);
    deneme.add(item9); deneme.add(item10);

    System.out.print("\nTree :"+deneme.toString());
}

```

Bu test metodu verilen sıra ile elemanları ekler ,eleman ekleme sonlandıktan sonra ağacı ekrana bastırır.

Test metodunun sonucu:

```

/usr/lib/jvm/java-1.8.0-openjdk-i386/bin/java ...

```

```

Tree :[40, 45, 30]
  [15, 70, 25]
    null
    [15, 80, 50]
      null
      [13, 80, 55]
        null
        null
    [70, 10, 15]
      null
      [69, 50, 35]
        [85, 50, 10]
          [68, 16, 30]
            null
            null
            null
          [66, 85, 45]
            null
            null

```

```

Process finished with exit code 0

```

- `find(ArrayList<E> target)`

```

@Test
void find() {
    add();
    System.out.print("Find result:"+deneme.find(item8));
}

```

Bu metod öncelikle ağacı oluşturur daha sonra bir elemanı find metoduna göndererek find metodunu test eder.

Test metodunun sonucu:

```

1 test passed - 99ms

null
[66, 85, 45]
null
null
Find result:[13, 80, 55]
Process finished with exit code 0

```

- **contains (ArrayList<E> target)**

```

@Test
void contains() {
    add();
    System.out.print(deneme.contains(item6));
}

```

Bu metod öncelikle ağacı oluşturur daha sonra bir elemanı contains metoduna göndererek contains metodunu test eder.

Test metodunun sonucu:

```

true
Process finished with exit code 0

```

- **delete (ArrayList<E> target)**

```

@Test
void delete() {
    add();
    deneme.delete(item3);
    System.out.println("Tree after deleting:"+deneme.toString());
}

```

Bu metod öncelikle ağacı oluşturur daha sonra bir elemanı delete metoduna göndererek delete metodunu test eder.

Test metodunun sonucu:

1 test passed - 137ms

```
[68, 10, 30]
Tree after deleting:[40, 45, 30]
  [15, 70, 25]
    null
    [15, 80, 50]
      null
      [13, 80, 55]
        null
        null
    [70, 10, 15]
      null
      [68, 16, 30]
        [85, 50, 10]
          null
          null
        [66, 85, 45]
          null
          null
```

Process finished with exit code 0

- `remove (ArrayList<E> target)`

```
@Test
void remove() {
    add();
    deneme.remove(item3);
    System.out.print("Tree after removing:"+deneme.toString());
}
```

Bu metod öncelikle ağacı oluşturur daha sonra bir elemanı remove metoduna göndererek remove metodunu test eder.

Test metodunun sonucu:

```

Tree after removing:[40, 45, 30]
[15, 70, 25]
  null
  [15, 80, 50]
    null
    [13, 80, 55]
      null
      null
[70, 10, 15]
  null
  [68, 16, 30]
    [85, 50, 10]
      null
      null
    [66, 85, 45]
      null
      null

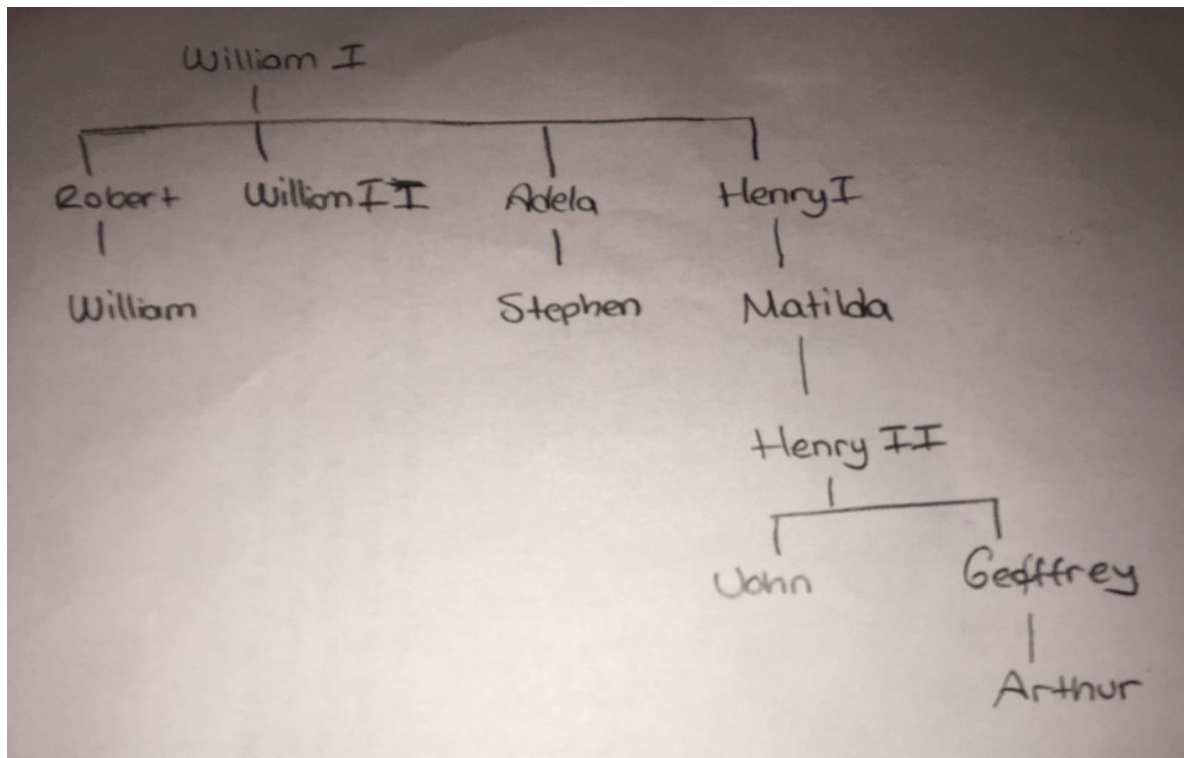
```

Process finished with exit code 0

## 2.2 Running Results

- **Part 1 Main Test İçin Kullanılan Ağaç**

Part 1 sınıfındaki metodlar test etmek için aşağıda görseli bulunan çok boyutlu ağaç kullanıldı.



Resim 2.2.1

- **Main metodu:**

```

package Part1;

public class MainTest {
    public static void main(String[] args) {

        Part1Tree deneme=new Part1Tree<String>();
        deneme.add("William I");
        deneme.add( parentitem: "William I", childrenitem: "Robert");
        deneme.add( parentitem: "Robert", childrenitem: "William");
        deneme.add( parentitem: "William I", childrenitem: "William II");
        deneme.add( parentitem: "William I", childrenitem: "Adela");
        deneme.add( parentitem: "William I", childrenitem: "Henry I");
        deneme.add( parentitem: "Adela", childrenitem: "Stephen");
        deneme.add( parentitem: "Henry I", childrenitem: "Matilda");
        deneme.add( parentitem: "Matilda", childrenitem: "Henry II");
        deneme.add( parentitem: "Henry II", childrenitem: "John");
        deneme.add( parentitem: "Henry II", childrenitem: "Geoffrey");
        deneme.add( parentitem: "Geoffrey", childrenitem: "Arthur");
        System.out.println( deneme.add( parentitem: "Al", childrenitem: "s"));
        System.out.println("ToString method uses PreOrderTraverse method:"+deneme.toString());

        System.out.println("\n\nLevel Order:\n");
        try {
            System.out.println("\nReturn Data:"+deneme.levelOrderSearch( target: "Arthur"));
        }catch (NullPointerException e)
        {
            System.out.println("null");
        }

        System.out.println("\n\nPost Order:\n");
        try {
            System.out.println("\n" + "Return Data:"+deneme.postOrderSearch( target: "William I"));
        }catch (NullPointerException e)
        {
            System.out.println("null");
        }
    }
}

```

#### ○ Main Metodunun Çıktısı:

```

MainTest
false
ToString method uses PreOrderTraverse method:
William I , Robert , William , William II , Adela , Stephen , Henry I , Matilda , Henry II , John , Geoffrey , Arthur ,

Level Order:

William I , Robert , William II , Adela , Henry I , William , Stephen , Matilda , Henry II , John , Geoffrey , Arthur ,
Return Data:Arthur

Post Order:

William , Robert , William II , Stephen , Adela , John , Arthur , Geoffrey , Henry II , Matilda , Henry I , William I ,
Return Data:William I

Process finished with exit code 0

```

- Part 2`deki main test metodu Part 2 için verilen testlere benzer şekilde çalışır.

## 2.3 Zaman karmaşıklığı(Time complexity)

- PART 1

- add(E item)

```
/**
 * This add method adds the first item into the tree.
 * @param firstItem Takes generic type parameter.
 * @return Returns boolean.
 */
public boolean add(final E firstItem)
{
    if (root!=null)
        return false;
    else{
        root=new Node<E>(firstItem);
        return true;
    }
}
```

Zaman karmaşıklığı sabittir.  $O(1)$

- add(E parentItem,E childItem)

```
/** @param parentItem
 * @param childItem
 * @return
 */
private Node<E> add(Node<E>localRoot,final E parentItem,final E childItem)
{
    if (localRoot==null)
    {
        addReturn=false;
        return null;
    }
    else if (parentItem.equals(localRoot.data))
    {
        if(localRoot.left==null)
            localRoot.left=new Node<E>(childItem);
        else
            traverseSibling(localRoot.left).right=new Node<E>(childItem);
        addReturn=true;
        return localRoot;
    }else
    {
        localRoot.right=add(localRoot.right,parentItem,childItem);
        localRoot.left=add(localRoot.left,parentItem,childItem);
        return localRoot;
    }
}
```

Zaman karmaşıklığı logaritmiktir. traverseSiblin metodu parametre olarak gönderilen düğümün kardeş sayısı kadar çalışır .(Kardeş sayısı m olsun). $O(m.\log n)$

- **levelOrderSearch(E target), postOrderSearch(E target)**

```
/**
 * Traverses the tree in level order.Return the Node reference if the item is in the tree and null if it is not in the tree.
 * <p>
 * <li>if localRoot equals the null
 * <li>That means target is not in the tree.The method returns null</li>
 * If target equals the data of the localRoot
 * <li>That means target is in the tree.The method returns localRoot</li>
 * Otherwise this method calls itself .
 * <li>this method adds the elements of the tree into the queue in level order.At the same time
 * when making the insert, the method search the target and deletes the element ,that made the comparison ,
 * from the tail.</li>
 * </p>
 * @param localRoot
 * @param target
 * @param queue
 * @return Returns the Node reference.
 */
protected Node<E> levelOrderSearch(Node localRoot,E target,Queue<Node<E>> queue) {
    if (localRoot==null)
        return null;
    System.out.print(localRoot+" , ");
    if (localRoot.data.equals(target))
        return localRoot;
    else
    {
        //the root is added into the queue
        if (localRoot==root)
            queue.add(root);
        //the child of the root is added in to the queue
        if (localRoot.left!=null)
        {
            //the first child of the localRoot is added into the queue
            queue.add(localRoot.left);
            Queue<Node<E>> que=new LinkedList<>();
            //the other child of the localRoot is added into the queue with the help of the addSibling method.
            addSibling(localRoot.left,que);
            queue.addAll(que);
        }
        //the head of queue is deleted
        queue.poll();
        //
        return levelOrderSearch(queue.element(),target,queue);
    }
}
```

Zaman karmaşıklığı lineerdir. addSibling metodu parametre olarak gönderilen düğümün kardeş sayısı kadar çalışır .(Kardeş sayısı m olsun).**O(m.n)**

- **preOrderTraverse(Node node, StringBuilder sb, int depth)**

```
/**
 * This traversal method visits the root first, then the left subtree and finally the right subtree.
 * @param node The local root
 * @param sb The string buffer to save the output
 * @param depth The depth
 */
@Override
protected void preOrderTraverse(Node node, StringBuilder sb, int depth) {
    if (node != null)
    {
        sb.append(node.data.toString()+" , ");
        preOrderTraverse(node.left,sb,depth);
        preOrderTraverse(node.right,sb, depth);
    }
}
```

Zaman karmaşıklığı lineerdir. **O(n)**

- Part 2
  - `add(ArrayList<E> item)`

```

*/
private Node<E> add(Node<E> localRoot, ArrayList<E> item, int level) {
    // item is not in the tree insert it.
    if (localRoot == null)
    {
        addReturn = true;
        dimension = item.size();
        return new Node<>(item);
    }
    if (localRoot.data.equals(item))
    {
        // item is equal to localRoot.data
        addReturn = false;
        return localRoot;
    }
    else
    {
        // Looks at the left if the element at the current size is smaller or equal than the root element.
        if (item.get(level % dimension).compareTo(localRoot.data.get(level % dimension)) <= 0)
            localRoot.left = add(localRoot.left, item, level: level + 1);
        else
            localRoot.right = add(localRoot.right, item, level: level + 1);
        return localRoot;
    }
}

/**
 * Determine if an item is in the tree

```

Zaman karmaşıklığı logaritmiktir. **O(logn)**

- `find(ArrayList<E> target)`

```

/**
 * Recursive find method.
 * @param target
 * @param localRoot
 * @param level
 * @return
 */
private Node<E> find(ArrayList<E> target, Node<E> localRoot, int level)
{
    if (localRoot == null)
        return localRoot;
    // Compare the target with the data field at the root.
    if (localRoot.data.equals(target))
        return localRoot;
    else if (target.get(level % dimension).compareTo(localRoot.data.get(level % dimension)) < 0)
        return find(target, localRoot.left, level: level + 1);
    else
        return find(target, localRoot.right, level: level + 1);
}

```

Zaman karmaşıklığı logaritmiktir. **O(logn)**

- `contains(ArrayList<E> target)`

```

/**
 * @Override
 * public boolean contains(ArrayList<E> target) {
 *     if (find(target, root, level: 0) == null)
 *         return false;
 *     return true;
 * }

```

Zaman karmaşıklığı logaritmiktir. **O(logn)**



- delete (ArrayList<E> target)

```

* @return
*/
private Node<E> delete(Node<E> localRoot, ArrayList<E> item, int level) {

    if (localRoot == null) {
        deleteReturn=null;
        return localRoot;
    }
    int compResult = item.get(level%dimension).compareTo(localRoot.data.get(level%dimension));
    if (compResult < 0 || (compResult==0 && !item.equals(localRoot.data))) {
        // item is smaller than localRoot.data.
        localRoot.left = delete(localRoot.left, item, level: level+1);
        return localRoot;
    }
    else if (compResult > 0) {
        // item is larger than localRoot.data.
        localRoot.right = delete(localRoot.right, item, level: level+1);
        return localRoot;
    }
    else {
        deleteReturn=localRoot.data;
        if (localRoot.left == null) {
            // If there is no left child, return right child
            // which can also be null.
            return localRoot.right;
        }
        else if (localRoot.right == null) {
            // If there is no right child, return left child.
            return localRoot.left;
        }
        else {
            deleteReturn=localRoot.data;
            if (isLeaf(localRoot.left)) {
                // 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 = null;
                return localRoot;
            } else {
                // Search for the inorder predecessor and
                // replace deleted node's data with ip.

                Node<E> max= findLargestChild(localRoot.left, Max: null, dim: level%dimension);
                localRoot= delete(localRoot, max.data, level: 0);
                localRoot.data = max.data;
                return localRoot;
            }
        }
    }
}
}
}

```

Zaman karmaşıklığı logaritmiktir. **O(logn)**

- remove (ArrayList<E> target)

```

/**
 *Removes target from tree.
 * @param target Item to be removed
 * @return
 */
@Override
public boolean remove(ArrayList<E> target) { return (delete(target)!=null); }

```

Zaman karmaşıklığı logaritmiktir. **O(logn)**