Consider that **BST<E>** is a class that represents a **generic binary search tree** and **Node<E>** is a nested class that represents a **node of that tree**, as shown below.

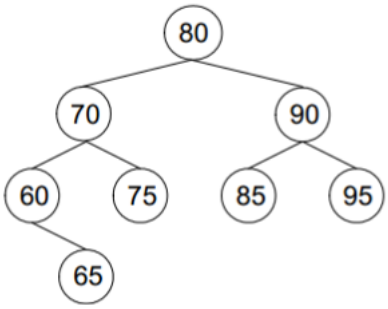
```
public class BST<E extends Comparable<E>> implements BSTInterface<E>
     protected Node<E> root = null;       // root of the tree
...
//-----------------------------------------------------------------
protected static class Node<E> {
  private E element;              // an element stored at this node
  private Node<E> left;          // a reference to the left child (if any)
  private Node<E> right;        // a reference to the right child (if any)
...
}
//-----------------------------------------------------------------
}
```

1. Implement the class TREE by inheriting from the BST, and add to it the following methods:

a) `public boolean contains(E element)` that verifies if an element is in the tree.

b) `public boolean isLeaf(E element)` that verifies if an element is in a leaf.

c) `public Iterable<E> ascdes()` - returns an iterable list with the elements of the left subtree in increasing order and the elements of the right subtree in descending order. According to the figure, the result is: 60, 65, 70, 75, 80, 95, 90, 85.



d) `public BST<E> autumnTree()` – returns a new binary search tree, identical to the original, but without the leaves.

e) `public int[] numNodesByLevel()` – returns an array with the number of nodes in each level of the tree, position 0 number of nodes at level 0, position 1 number of nodes at level 1, …

f) Make a function that indicates whether a TREE is a perfectly balanced tree, i.e., a tree in which all nodes have balance factor = 0.

2. Insert the following sequence of nodes: A, Z, B, C, X, D, Q, E, V, M, Q in AVL tree and draw the tree after each insertion, showing the rotations made.

3. Remove B, A, F, P, C elements and draw the state of the tree after each deletion, with the rotations made.