A BST is a binary tree where each node has a Comparable element and satisfies the restriction that the element in any node is greater than the elements in all nodes in that node's left subtree and lesser than the elements in all nodes in that node's right subtree.

Each node contains an element, a left link, and a right link. The left link points to a BST for items with lesser elements, and the right link points to a BST for items with greater elements, as depicted in the Figure 1.

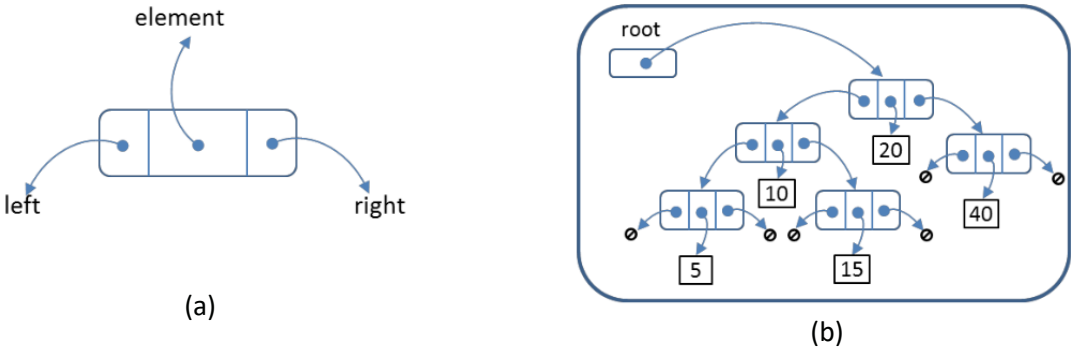

Figure 1 - (a) A single node; (b) binary tree

Download the project PL6_BST_for_students from moodle. The project contains one generic source outer class BST<E>, which has one nested class Node<E> (already complete) and a text file (xxx.xxx). Three test classes BSTTest, TREETest and TREE_WORDSTest.

Exercise 1: Develop the generic class BST<E> with the following public interface:

```
BST(): the BST's constructor.
boolean isEmpty(): returns true if the tree is empty, false otherwise.
int size(): returns the number of elements in the tree
void insert(E element): inserts (or replaces if already exists) an element in the tree
void remove(E element): removes an element (node) from the tree.
int height(): returns the height of the tree
E smallestElement(): returns the smallest element within the tree.
Iterable<E> inOrder(): returns the sequence of elements resulting from an in-order
traversal of the tree
Iterable<E> preOrder(): returns the sequence of elements resulting from a pre-order
traversal
Iterable<E> postOrder(): returns the sequence of elements resulting from an post-order
traversal
Map<Integer,List<E>> nodesByLevel(): returns a map containing for each tree level
a list of nodes that belong to that level
```

a) Implement all the methods of the BST's interface.

b) Test your implementation running the tests in the BSTTest class.

Exercise 2:

Implement a binary search tree from the words of a text file. This tree must store information on the number of occurrences of each word. Admit as valid separators the characters: comma, period, space and line break.

a) Implement the class TREE_WORDS to represent a binary search tree with elements of the type TextWord;

b) Implement the necessary adjustments in a way that when a word is inserted into the tree the number of its occurrences is updated;

c) Implement the public method **`Map<Integer, List<String>> getWordsOcurrences()`** that should return a map containing, for each number of occurrences found, the list of words that have the same number of occurrences.

Exercise 3:

Implement a generic method that sorts an unordered List<E>.

Exercise 4:

As you know, an ***AVL tree*** is a binary search tree where each node is balanced in height, that is, where the difference between the height of its right and left subtrees are not more than 1. The aim of this worksheet is to make an implementation of the ***AVL tree*** based on the ***BST class***.

As with the ***BST*** class, the ***AVL*** class uses the generic parameter ***E*** to designate the element type stored at the nodes. The generic class AVL< E> extends the BST class and includes the following methods:

```
public class AVL <E extends Comparable<E>> extends BST<E> {
     int balanceFactor(Node<E> node);
     Node<E> rightRotation(Node<E> node);
     Node<E> leftRotation(Node<E> node);
     Node<E> twoRotations(Node<E> node);
     Node<E> balanceNode(Node<E> node);
     void insert(E element);
     void remove(E element);
}
```

a) Complete the generic class AVL< E> implementing all the methods above.

b) Test your implementation running the tests in the AVLTest class.