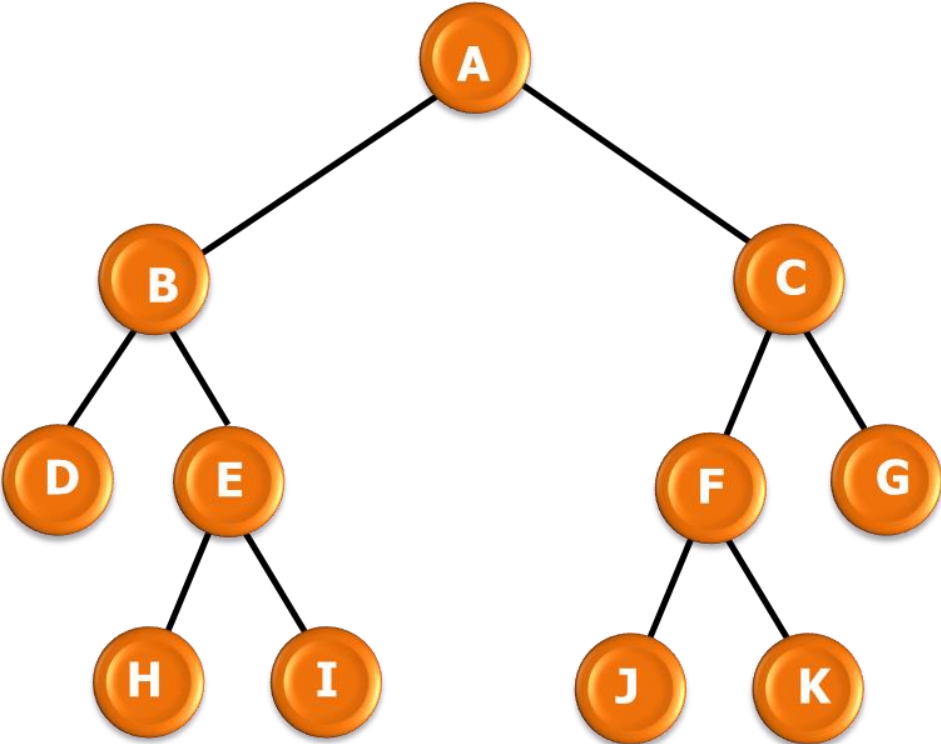


<b>TAD BST</b>	
 <pre> graph TD     A((A)) --- B((B))     A --- C((C))     B --- D((D))     B --- E((E))     E --- H((H))     E --- I((I))     C --- F((F))     C --- G((G))     F --- J((J))     F --- K((K)) </pre>	
<p>{inv: The following order property must be met. (1) All the data in its left subtree is lessor equal to the data that the root occupies. (2) All nodes in its right subtree is greater than the data which occupies the root. (3) The left and right are also BST}</p>	
<p>BST: &lt;&gt; -----&gt; &lt;BST&gt; <b>Constructor</b>  insertE: &lt; key,value &gt; -----&gt; &lt;&gt; <b>Modifier</b>  insertE: &lt;node&gt; -----&gt; &lt;&gt; <b>Modifier</b>  insertE: &lt;node1,node2&gt; -----&gt; &lt;&gt; <b>Modifier</b>  searchEquals: &lt;key&gt; -----&gt; &lt;ArrayList&lt;V&gt;v&gt;<b>Analyzer</b>  searchEquals: &lt;node, key&gt; -----&gt; &lt;node&gt; <b>Analyzer</b>  inOrderLess: &lt;node, key&gt; -----&gt; &lt;&gt;<b>Analyzer</b>  inOrderMore: &lt;node, key&gt; -----&gt; &lt;&gt;<b>Analyzer</b>  indices: &lt;&gt; -----&gt; &lt; ArrayList&lt;V&gt;v&gt; <b>Modifier</b></p>	
BST()	
Create a BST empty	
{pre: }	
{post: BST b = $\emptyset$ }	
InsertE(key,value)	

Create a new node to insert to the tree
---

{pre: key and value non-null}
-------------------------------

{post: Node created}
----------------------

InsertE(Node<K,V>)
--------------------

Insert a new node to the tree
-------------------------------

{pre: node non-null}
----------------------

{post: The new node is added to the tree}
---

InsertE(Node<K,V> current, Node<K,V> newNode)
---

Insert a new node to the tree
-------------------------------

{pre: current and newNode non-null}
-------------------------------------

{post: The new node is added to the tree}
---

searchEquals(Node<K,V> current, K key)
--

Search for a specific node according to the key
---

{pre: current!=null}
----------------------

{post: node if node.getKey=key or null if the opposite happens }
--

searchEquals (K key)
----------------------

Returns a list with the indices of the found key
--

{pre: }
---------

{post: ArrayList<V> v if v≠∅ or null if the opposite happens }
--

inOrderLess (Node<K,V> node, K key)
-------------------------------------

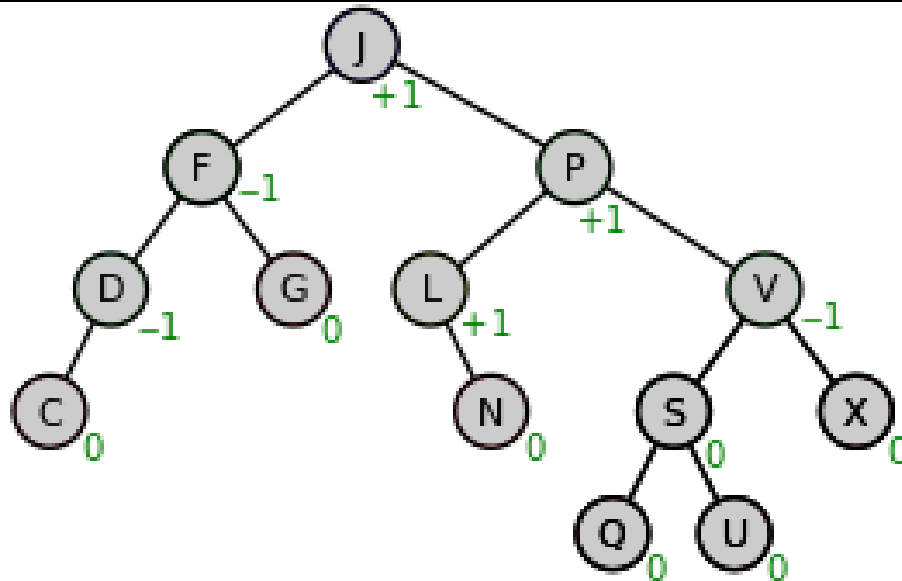
Find all nodes with less than the value of the key
--

{pre: node!=null}
{post: }

inOrderMore (Node<K,V> node, K key)
Find all nodes with more than the value of the key
{pre: node!=null }
{post: }

indices()
Returns a list with all the values of the list of nodes found
{pre: }
{post: ArrayList<V> v if v≠∅ or null if the opposite happens }

## TAD AVL



{inv:  $BF(node) = Height(RightSubtree(node)) - Height(LeftSubtree(node)) \mid BF(node) \in \{-1, 0, 1\}$ }

AVL:  $\langle \rangle \rightarrow \langle AVL \rangle$  **Constructor**

insertE:  $\langle key, value \rangle \rightarrow \langle \rangle$  **Modifier**

insertE:  $\langle node \rangle \rightarrow \langle \rangle$  **Modifier**

insertE:  $\langle node1, node2 \rangle \rightarrow \langle \rangle$  **Modifier**

searchEquals:  $\langle key \rangle \rightarrow \langle ArrayList<V>v \rangle$  **Analyzer**

searchEquals:  $\langle node, key \rangle \rightarrow \langle node \rangle$  **Analyzer**

inOrderLess:  $\langle node, key \rangle \rightarrow \langle \rangle$  **Analyzer**

inOrderMore:  $\langle node, key \rangle \rightarrow \langle \rangle$  **Analyzer**

indices:  $\langle \rangle \rightarrow \langle ArrayList<V>v \rangle$  **Modifier**

rotateL:  $\langle node \rangle \rightarrow \langle NodeAVL<K, V> \rangle$  **Modifier**

rotateR:  $\langle node \rangle \rightarrow \langle NodeAVL<K, V> \rangle$  **Modifier**

balance:  $\langle node \rangle \rightarrow \langle NodeAVL<K, V> \rangle$  **Modifier**

Insert:  $\langle key, value \rangle \rightarrow \langle boolean \rangle$  **Modifier**

AVL()

Create a AVL empty

{pre: right child height minus left child height is the roll factor to be balanced}

{post: AVL that it is either empty, or both children they are also AVL and the difference between their heights is less than or equal to 1}

InsertE(key,value)

Create a new node to insert to the tree
---

{pre: key and value non-null}
-------------------------------

{post: Node created}
----------------------

InsertE(Node<K,V>)
--------------------

Insert a new node to the tree
-------------------------------

{pre: node non-null}
----------------------

{post: The new node is added to the tree}
---

InsertE(Node<K,V> current, Node<K,V> newNode)
---

Insert a new node to the tree
-------------------------------

{pre: current and newNode non-null}
-------------------------------------

{post: The new node is added to the tree}
---

searchEquals(Node<K,V> current, K key)
--

Search for a specific node according to the key
---

{pre: current!=null}
----------------------

{post: node if node.getKey=key or null if the opposite happens }
--

searchEquals (K key)
----------------------

Returns a list with the indices of the found key
--

{pre:tree!=null }
-------------------

{post: ArrayList<V> v if v≠∅ or null if the opposite happens }
--

inOrderLess (Node<K,V> node, K key)
-------------------------------------

Find all nodes with less than the value of the key
--

{pre: node!=null}
-------------------

{post: }
----------

inOrderMore (Node<K,V> node, K key)
-------------------------------------

Find all nodes with more than the value of the key
--

{pre: node!=null }
--------------------

{post: }
----------

indices()
-----------

Returns a list with all the values of the list of nodes found
---

{pre:tree!=null }
-------------------

{post: ArrayList<V> v if v≠∅ or null if the opposite happens }
--

rotateL(NodeAVL<K,V> node)
----------------------------

Rotate the tree to left for balance the tree
--

{pre: AVL!=null, (is not_Empty(AVL)) and (is not_Empty(Hijo_Left) }
---

{post: The three rotate to the left }
---------------------------------------

rotateR(NodeAVL<K,V> node)
----------------------------

Rotate the tree to right for balance the tree
---

{pre: AVL!=null, (is not_Empty(AVL)) and (is not_Empty(Hijo_Right) }
--

{post: The three rotate to the right }
--

balance(NodeAVL<K,V> node)
----------------------------

Balance the tree when the child has a height h + 2
--

{pre: the right or left child has an h + 2 }
--

{post:the h of tree are = h }
-------------------------------

insert (K key, V value)
-------------------------

Insert a new node to the tree
{pre: key and value non-null}
{post: The new node is added to the tree}

TAD RBT
<p>{inv: The following order property must be met. (1) Every node has a color either red or black.(2) The root of the tree is always black.(3) There are no two adjacent red nodes (A red node cannot have a red parent or red child).(4) Every path from a node (including root) to any of its descendant's NULL nodes has the same number of black nodes. }</p>
<p>RBT: &lt;&gt; -----&gt; &lt;RBT&gt; <b>Constructor</b>  insertE: &lt; key,value &gt; -----&gt; &lt;&gt; <b>Modifier</b>  insertE: &lt;node&gt; -----&gt; &lt;&gt; <b>Modifier</b>  insertE: &lt;node1,node2&gt; -----&gt; &lt;&gt; <b>Modifier</b>  searchEquals: &lt;key&gt; -----&gt; &lt;ArrayList&lt;V&gt;v&gt;<b>Analyzer</b>  searchEquals: &lt;node, key&gt; -----&gt; &lt;node&gt; <b>Analyzer</b>  inOrderLess: &lt;node, key&gt; -----&gt; &lt;&gt;<b>Analyzer</b>  inOrderMore: &lt;node, key&gt; -----&gt; &lt;&gt;<b>Analyzer</b>  indices: &lt;&gt; -----&gt; &lt; ArrayList&lt;V&gt;v&gt; <b>Modifier</b>  rotateLeft:&lt;node&gt; -----&gt; &lt; NodeRBT&lt;K,V&gt; node&gt; <b>Modifier</b>  rotateRight:&lt;node&gt; -----&gt; &lt; NodeRBT&lt;K,V&gt; node&gt; <b>Modifier</b>  insertNode:&lt;key,value&gt; -----&gt; &lt;boolean&gt; <b>Modifier</b>  InsertF:&lt;node&gt; -----&gt; &lt;&gt; <b>Modifier</b></p>

RBT()
Create a RBT empty
{pre: }
{post: RBT b = $\emptyset$ }



InsertE(key,value)
Create a new node to insert to the tree
{pre: key and value non-null}
{post: Node created}

InsertE(Node<K,V>)
Insert a new node to the tree
{pre: node non-null}
{post: The new node is added to the tree}

InsertE(Node<K,V> current, Node<K,V> newNode)
Insert a new node to the tree
{pre: current and newNode non-null}
{post: The new node is added to the tree}

searchEquals(Node<K,V> current, K key)
Search for a specific node according to the key
{pre: current!=null}
{post: node if node.getKey=key or null if the opposite happens }

searchEquals (K key)
Returns a list with the indices of the found key
{pre: }
{post: ArrayList<V> v if v≠∅ or null if the opposite happens }

inOrderLess (Node<K,V> node, K key)
Find all nodes with less than the value of the key

{pre: node!=null}
{post: }

inOrderMore (Node<K,V> node, K key)
Find all nodes with more than the value of the key
{pre: node!=null }
{post: }

indices()
Returns a list with all the values of the list of nodes found
{pre: tree!=null}
{post: ArrayList<V> v if v≠∅ or null if the opposite happens }

rotateLeft(NodeAVL<K,V> node)
Rotate the tree to left for balance the tree
{pre: RBT !=null, (is not_Empty(RBT)) and (is not_Empty(Hijo_Left))}
{post: The three rotate to the right}

rotateRight (NodeAVL<K,V> node)
Rotate the tree to right for balance the tree
{pre: RBT !=null, (is not_Empty(RBT)) and (is not_Empty(Hijo_Right))}
{post: The three rotate to the right}

insertNode(K k,V v)
Create a new node to insert to the tree
{pre: key and value non-null}
{post: Node created}

insertF(NodeRBT<K,V> node)
Insert a new node to the tree
{pre: node non-null}
{post: The new node is added to the tree}