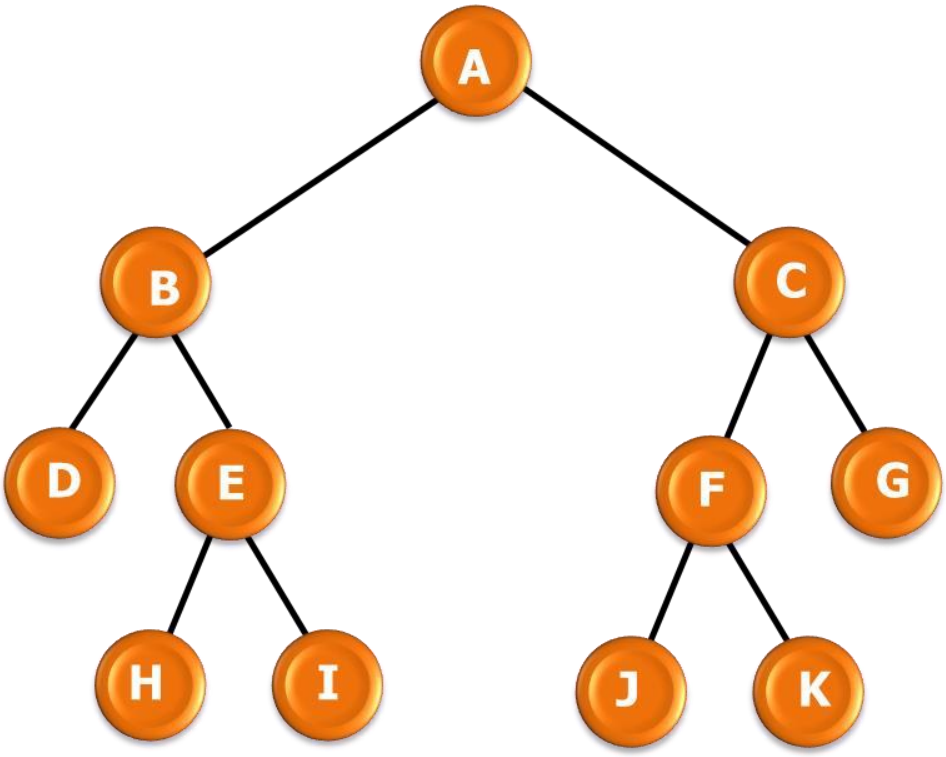
BST ADT



{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 are greater than the data which occupies the root.
(3) The left and right are also BST} }

Operations:

▪ BST (constructor)	-	→ BST
▪ insert (modifier)	Value	→ BST
▪ insert (modifier)	Node	→ BST
▪ search (analyzer)	Value	→ Value
▪ successor (analyzer)	Node	→ Node
▪ delete (modifier)	Value	→ Node
▪ inOrderLess (analyzer)	Node x Value	→ Void
▪ inOrderMore (analyzer)	Node x Value	→ Void
▪ searchEquals (analyzer)	Node x Value	→ Void
▪ eraseNodes (modifier)	-	→ Void

BST (-)
"Builds an empty binary search tree" {pre: - } {post: BST b = ∅}

insert (Value)
"Insert a node with the desired value" {pre: BST b and a value} {post: a BST b with a new node added}

insert (Node)
"Insert a new node to the binary search tree" {pre: BST b} {post: a binary search tree with a new node added with the given order}

search (Value)
“Search the value in the nodes of the binary search tree” {pre: BST ≠ ∅} {post: Node if the value of the value equals the found node, null otherwise}

successor (Node)
"Determines the node successor" {pre: BST ≠ ∅} {post: The node successor of the given node if exists, null otherwise}

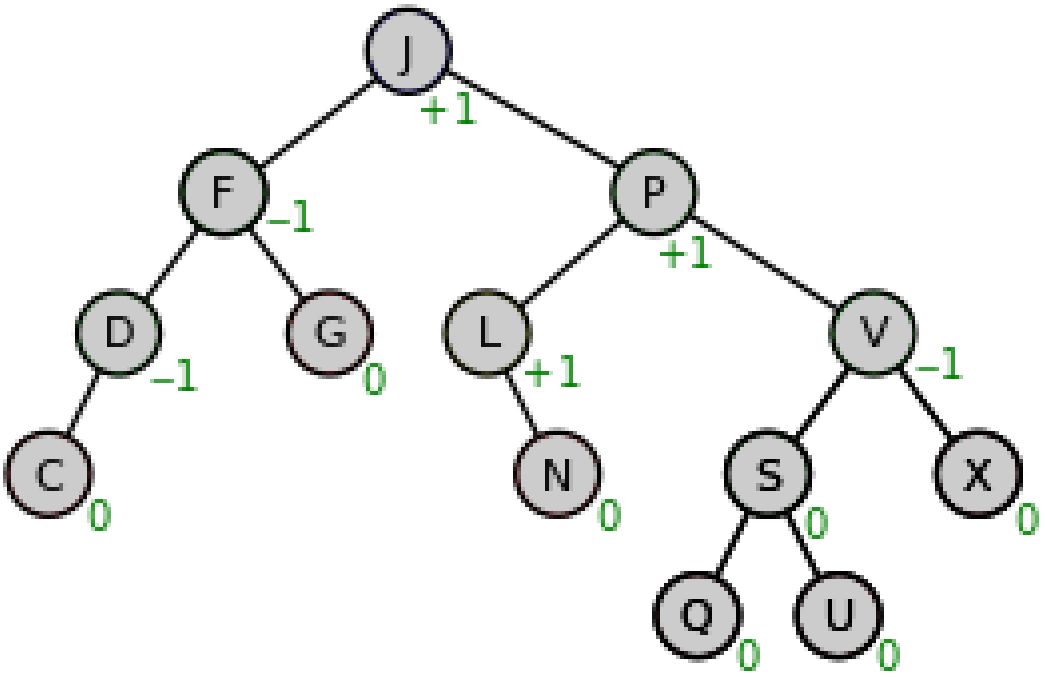
delete (Value)
"Delete the node corresponding the given value" {pre: BST ≠ ∅} {post: the deleted node}

inOrderLess **(Node, Value)**
"Fulfill a list with all the values lesser than the given value" {pre: BST ≠ ∅} {post: -}

inOrderMore **(Node, Value)**
"Fulfill a list with all the values greater than the given value" {pre: BST ≠ ∅} {post: -}

searchEquals **(Node, Value)**
"Search all the values exactly equal to the given value and saves the nodes in a list" {pre: a node and a value ≠ null} {post: -}

eraseNodes (-)
"Delete all the nodes stored in the list" {pre: BST and a list ≠ ∅} {post: - }

AVL ADT

{inv: BF (node) = Height(RightSubtree(node)) - Height(LeftSubtree(node)) | BF (node) € {- 1,0,1}}
Operations:

▪ AVL (constructor)	-	→ AVL
▪ rotateL (modifier)	Node	→ Node
▪ rotateR (modifier)	Node	→ Node
▪ balance (modifier)	Node	→ Node
▪ insert (modifier)	Key	→ Node

AVL (-)
"Builds an empty AVL BST" {pre: - } {post: AVL a = ∅}

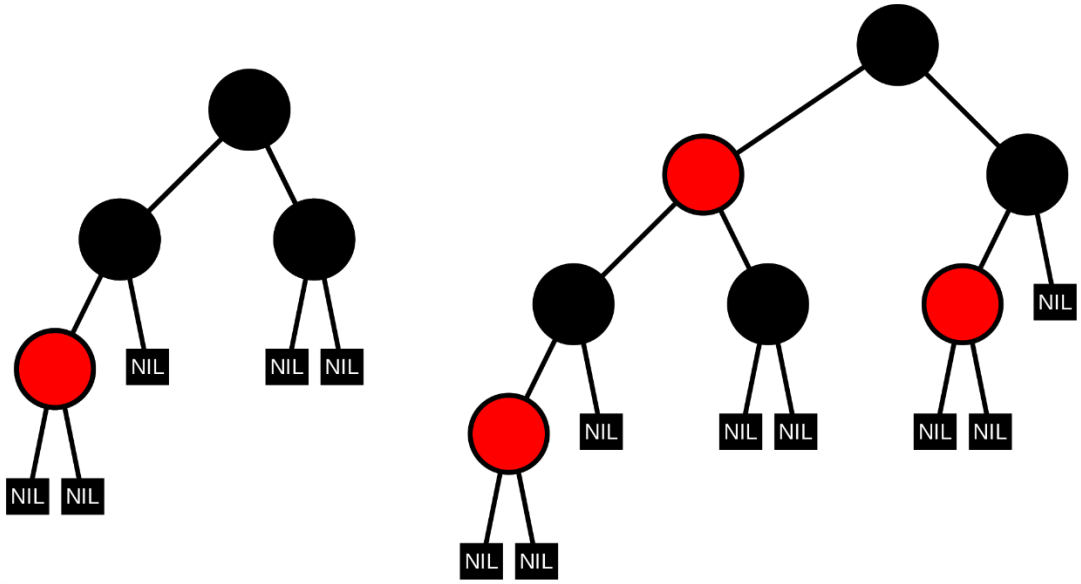
rotateL (Node)
"Rotate the tree or a subtree of the AVL tree to the left" {pre: an AVL a ≠ ∅ and a node different from null} {post: a node }

rotateR (Node)
" Rotate the tree or a subtree of the AVL tree to the right" {pre: an AVL a ≠ ∅ and a node different from null } {post: a node}

balance **(Node)**
“Balance the tree or a subtree of the AVL tree given the balance factor” {pre: an AVL a ≠ ∅ and a node different from null } {post: returns the root of the new balanced tree}

insert (key)
"Insert a node with the key while balancing the tree in the process keeping the balance factor between -1, 0 or 1 " {pre: key different from null} {post: a tree with a new node with the key}

RBT ADT



{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. }

Operations:

▪ RBT (constructor)	-	→ RBT
▪ rotateRight (modifier)	Node	→ Node
▪ rotateLeft (modifier)	Node	→ Node
▪ insertNode (modifier)	K	→ Boolean
▪ insertF (modifier)	Node	→ Void

RBT **(-)**

"Builds an empty red-black tree"

{pre: - }

{post: RBT r = ∅}

rotateRight **(Node)**
"Rotate the tree or the subtree of the RBT to the right to keep the rbt balanced"
{pre: RBT r ≠ ∅ and a node right child ≠ ∅ }
{post: that tree or subtree rotated to the right]

rotateLeft **(Node)**
" Rotate the tree or the subtree of the RBT to the left to keep the rbt balanced "
{pre: RBT r ≠ ∅ and a node left child ≠ ∅ }
{post: that tree or subtree rotated to the left}

insertNode **(K)**
“Create a node with the given key k and insert it to the RBT”
{pre: k r ≠ ∅}
{post: true if was inserted, false otherwise}

insertF **(Node)**
"Insert a new node to the tree"
{pre: node ≠ ∅ }
{post: the node inserted in the tree with the given order and the tree balanced