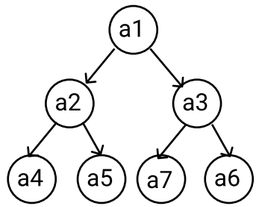
ADT AVL TREE

BST TREE = {a1, a2, a3 ... aN}

- a1 is the main element, a2 and a3 are subtrees of a1, any element less than a1 goes to the left, and any element to the right is greater than a1.

-The AVL tree element has a balance factor calculated by depth of its sub trees



a4<a2<a5<a1<a6<a3<a7

Balance factor: maxDepthRightSide-maxDepthLeftSide
Inv: {a1>a2, a1<a3} && |BalanceFactor|= 1 or 0 for any BST tree and sub tree, to the left of the element, elements are less than and to right of the element, elements are greater than. The height of the left branch can't be more than one unit than the right branch or viceversa.

Primitive operations:

-createAVL: ->AVL
-Insert: Element x AVL ->AVL
-delete: Element x AVL ->AVL
-search: AVL ->Element
-searchElement: AVL ->Element
-rotateRight: Element x AVL -> AVL
-rotateLeft: Element x AVL -> AVL
-rebalance: Element x AVL -> AVL
-RecalculateFactorBalances: Element x AVL -> AVL
-maxDepth: Element x AVL -> AVL

Insert(K key,E newItem) : Modifier

"Insert a new key inside the binary tree, if the key already exists, insert a new position"

{ pre: AVL Binary Tree initializated }
{ post: Increments the depth of the branch with +1 in this specific sub-tree }

Delete(K key): Modifier

"Delete a specific element or key from the binary tree"

{ pre: AVL Binary Tree initializated }
{ post: Decrements the depth of the branch with -1 in this specific sub-tree }

Search(K key): Analayzer

"Search a specific key value inside the Binary Tree and returns it"

{ pre: AVL Binary Tree initializated }
{ post: Return the ArrayList of elements or return a "False" if the the key don't exists }

SearchElement(K key): Analyzer

"Search a specific element with a unique key value and returns it"

{ pre: AVL Binary Tree initializated }
{ post: Element : The element with the specific key value, if the element don't exists, it returns False }

RotateRight(Element) : Modifier

"Dequeue the last element from the Queue and delete it"

{ pre: AVL Binary Tree must initializated and the rotate objectives must exists != null }
{ post: Binary three structure modified with a Right rotation }

RotateLeft(Element) : Modifier

"Return the total length of the queue in a integer variable"

{ pre: AVL Binary Tree must initializated and the rotate objectives must exists != null }
{ post: Binary three structure modified with a Left rotation }

Rebalance( ) : Modifier

"Rebalance the Binary Tree to secure the efficiency factor"

{ pre: AVL Binary Tree must initializated and unbalanced }
{ post: It determinates the rotation case with a Switch and call the respective rotations }

CreateAVL( ) : Constructor

"Create (Initializate) a new empty AVL Binary tree to add new elements"

{ pre: TRUE }
{ post: NewTree: The new created AVL Binary tree ready to add new elements }

RecalculateFactorBalances(Element) : Modifier

"Recalculate the new factor balances of all father nodes up the inserted element"

{ pre: AVL Binary Tree must stay initializated }
{ post: AVL Binary Tree with the factor balances recalculated }

MaxDepth(Element) : Modifier

"Calculate the deepest branch"

{ pre: AVL Tree must stay initializated }
{ post: Integer of the max deep of the branch}