BinarySearchTree

TAD BinaryTree
BinaryTree = {Node = <Node>}
{**inv:** BinaryTree.root > BinaryTree.root.getLeft() && BinaryTree.root <= BinaryTree.root.getRight()}
Primitive operations:
TreeNode = Element of the BinaryTree

Methods	Operation type	Input	Output
BinaryTree	Constructor		→ BinaryTree
createNode	Constructor	T (value)	→TreeNode
insertNode	Modifier	TreeNode x TreeNode	
isEmpty	Analyzer		→ Boolean
isLeaf	Analyzer	TreeNode	→ Boolean
getTreeNode	Analyzer	TreeNode x T (value)	→ TreeNode
deleteNode	Modifier	TreeNode x T (value)	→ TreeNode
successor	Analyzer	TreeNode	→ T
predecessor	Analyzer	TreeNode	→ T

BinaryTree():
*Create an empty BinaryTree. *
{pre:}
{post: BinaryTree was created}

createNode():
*Create a new TreeNode. *
{pre: TRUE}
{post: TreeNode created. If the BinaryTree is empty, then the root of the BinaryTree is n}

insertNode(current, newNode):
*It's a recursive method to insert the newNode. If the newNode is less than the current, then, the newNode will be inserted at the left of the current (It will become the left of the current). But, if the newNode is greater than the current, then, the newNode will be inserted at the right of the current (It will become the right of the current. *
{pre: BinaryTree has to be initialized} {pre: newNode must to exist (It must be different from null because we need to compare somehow)}
{post: The height of the BinaryTree increases one}

isEmpty():
*Verify if the BinaryTree has elements (TreeNodes) or not. *
{pre: BinaryTree must to exist}
{post: `True` if root == null `False` otherwise}

isLeaf():
*Verify if a TreeNode has left TreeNode, right TreeNode or both. *
{pre: TreeNode must to exist} {pre: TreeNode must to be in the BinaryTree}
{post: `True` if TreeNode.getLeft() == null && TreeNode.getRight() == null `False` otherwise}

getTreeNode(current, searchNode):
* It's a recursive method to get a TreeNode from the BinaryTree without deleting it. The TreeNode searchNode is obtained from the BinaryTree as long as it exists. *
{pre: BinaryTree must to be initialized} {pre: searchNode must to exist (It must be different from null because we need to compare somehow} {pre: searchNode exists in the BinaryTree}
{post: Returns the TreeNode to get}

deleteNode(root, delete) :

*It’s a recursive method to remove a TreeNode. The new element in the position of the TreeNode removed will be the successor or predecessor of the TreeNode removed. *
{pre: BinaryTree must to be initialized} {pre: delete must to exist (It must be different from null because we need to compare somehow)} {pre: delete exists in the BinaryTree}
{post: TreeNode returned and removed}

successor(root) :
* The successor is the smallest value present in the right subtree. This is a recursive method to find the successor of a TreeNode if and only if the root has a right child. If the root hasn’t a right child, we will look for a predecessor. *
{pre: BinaryTree must to be initialized} {pre: root must to exist} {pre: root exists in the BinaryTree}
{post: Value of the successor TreeNode is returned}

predecessor(root) :
* The predecessor is the greatest value present in the left subtree. This is a recursive method to find the predecessor of a TreeNode if and only if the root has a left child. If the root hasn’t a left child, we will look for a successor. *
{pre: BinaryTree must to be initialized} {pre: root must to exist} {pre: root exists in the BinaryTree}
{post: Value of the predecessor TreeNode is returned}