**AVL Tree Implementation - Documentation**

**AVL tree** (named after inventors **A**delson-**V**elsky and **L**andis) is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one. If at any time they differ by more than one, rebalancing is done to restore this property, by performing specific tree rotations.

In our implementation, each node of the tree has a value (info) of string type, and a key (key) which is a natural number. All the keys are distinct, and the order on the nodes of the tree refers only to the keys.

**Operations:**

**empty() -** the function returns a TRUE value if and only if the tree is empty.

**search(int k) -** the function searches for a node with the key k. If such node exists, it returns the value stored for it, otherwise it returns null.

**insert(int k, String s) -** inserting an element with value s and key k into the tree, if it does not exist. The function returns the total number of balancing operations that were required while repairing the tree in order to complete the operation (LR and RL rotations count as 2 balancing operations). If there is a node with key k in the tree, the function returns -1 and no insertion is performed.

**delete(int k) -** deleting a node with the key k in the tree, if exists. The function returns the number of balancing operations required while repairing the tree in order to complete the operation. If there is no node with the key k in the tree, the function returns -1.

**min() -** returns the value (info) of the node in the tree with the minimum key, or null if the tree is empty.

**max() -** returns the value (info) of the node in the tree with the maximum key, or null if the tree is empty.

**keysToArray() -** the function returns a sorted array containing all the keys in the tree, or an empty array if the tree is empty.

**infoToArray() -** the function returns a string array containing all the strings in the tree, sorted by keys. That is, the j-th value in the array is the string corresponding to the key that will appear in the j-th position in the output array of the keysToArray() function. This function also returns an empty array if the tree is empty.

**size() -** the function returns the number of nodes in the tree.

**split(int x) -** the function receives a key x that is found in the tree. The function separates the tree into 2 AVL trees where the keys of one tree are greater than x, and the keys of the other tree are less than x. The function is implemented with O(log⁡n) complexity.

**join(node x, tree t) -** the function receives a node x and an AVL tree t, all of whose keys are smaller (or larger) than the keys of the tree the function operates on. The function merges x and t into that tree. The operation runs in O(log⁡n) time. The function returns the cost of the join operation (difference of tree heights + 1).

**getRoot() -** returns the root of the tree.

**Implementation:**

**IAVLNode interface:**

This interface represents a general tree node, with basic operations to be performed on it.

In our implementation each leaf node in the tree has 2 "virtual" sons, that is, nodes without a key. A virtual node is not a real tree node, but an extension for convenience (it will make it easier to implement rotations in the balancing process since each node will have 2 sons).

The interface is needed for representing both real nodes and virtual nodes.

**Operations:**

**getKey() -** returns node's key (-1 for virtual nodes).

**getValue() -** returns node's value (info), for a virtual node return null.

**setLeft(IAVLNode node)** - sets the given node as the left child.

**getLeft() -** returns left child, if there is no left child returns null.

**setRight(IAVLNode node) -** sets the given node as the right child.

**getRight() -** returns right child, if there is no right child returns null.

**setParent(IAVLNode node) -** sets the given node as the parent.

**getParent() -** returns the parent, if there is no parent returns null.

**setHeight(int height)** - sets the height of the node.

**getHeight() -** returns the height of the node (-1 for virtual nodes).

**getSize() -** returns the size of the subtree of the node.

**isRealNode() -** returns True if the node is a non-virtual AVL node.

**updateSize() -** updates the size of the subtree of the node.

**update(IAVLNode node1, IAVLNode node2) -** sets node1 and node2 as the node's sons, and sets the node as their parent.

**AVLNode class:**

This class implements the IAVLNode interface. AVLNode object represents a tree node.

Each AVLNode object (a tree node) has the following fields:

1. **info -** type String, stores the value of the node.

2. **key -** type int, stores the key of the node.

3. **height -** type int, stores the height of the node.

4. **left -** type AVLNode, stores the left child of the node.

5. **right -** type AVLNode, stores the right child of the node.

6. **parent -** type AVLNode, stores the parent of the node.

7. **size -** type int, stores the size of the subtree that its root is the node.

Each AVLNode object has the following functions:

**Constructor:**

**public** AVLNode(String info, **int** key, AVLNode left, AVLNode right, AVLNode parent)

O(1) runtime complexity. Access to fields in constant time.

The function constructs an AVLNode object and returns it, with the fields initialized according to the received parameters. If the received key is -1, then it is a virtual node, therefore its height is set to -1, and the size of the subtree that the node is its root is set to 0.

**Default constructor:** **public** AVLNode()

O(1) runtime complexity. Access to fields in constant time.

The function constructs an object of type AVLNode and returns it, where the object represents a virtual node.

***get* and *set* functions:**

**public** **void** setKey(**int** k)  **public** **int** getKey()

**public** **void** setValue(String s)  **public** String getValue()

**public** **void** setLeft(IAVLNode node) **public** IAVLNode getLeft()

**public** **void** setRight(IAVLNode node) **public** IAVLNode getRight()

**public** **void** setParent(IAVLNode node) **public** IAVLNode getParent()

**public** **void** setHeight(**int** height) **public** **int** getHeight()

**public** **int** getSize()

O(1) runtime complexity. Access to fields in constant time.

These functions are defined for each field. The *get* function returns the field value, and the *set* function updates the field to be the received parameter. The *set* function of the *info* field is called *setValue* and its *get* function is called *getValue*. The *size* field does not have a *set* function.

**Function:** **public** **boolean** isRealNode()

O(1) runtime complexity. Tests that take constant time.

This function checks whether the node is a real (not virtual) node.

The function returns True iff the node is real (i.e. its height is not -1).

**Helper function:** **public** **void** updateSize()

O(1) runtime complexity. Performing actions that take constant time.

The function updates the size of the node, according to the size of its sons, assuming that it is a real node. Otherwise, its size is updated to 0.

**Helper function:** **public** AVLNode min()

O(log(n)) runtime complexity. In the worst case, the loop is executed O(log(n)) times (as the height of the tree) and the operations inside the loop are executed in a constant time.

This function has a pre condition: the node on which the function operates must be real.

The function returns the node with the minimum key in the subtree of the node. The function advances to the left child as long as it is real and this way reaches the minimum node in the subtree.

**Helper function:** **public** AVLNode max()

O(log(n)) runtime complexity. In the worst case, the loop is executed O(log(n)) times (as the height of the tree) and the operations inside the loop are executed in a constant time.

This function has a pre condition: the node on which the function operates must be real.

The function returns the node with the maximum key in the subtree of the node. The function advances to the right child as long as it is real and this way reaches the maximum node in the subtree.

**Helper function:** **public** AVLNode successor()

O(log(n)) runtime complexity. In the case where the node *this* has a right child, then its successor is the node with the minimum key in the right subtree of *this*. We find it by using min(), which costs O(log(n)). In the case that *this* does not have a right child, we search for the successor in the part above the node until we reach a situation where we are in a left subtree of some node, which means the complexity is also O(log(n)) since we could search along the entire tree.

This function has a pre condition: the node on which the function operates must be real.

The function returns the node with the successoring key in the tree to *this* node's key.

**Helper function:** **public** AVLNode predecessor()

O(log(n)) runtime complexity. In the case where the node *this* has a left child, then its predecessor is the node with the maximum key in the left subtree of *this*. We find it by using max(), which costs O(log(n)). In the case that *this* does not have a left child, we search for the predecessor in the part above the node until we reach a situation where we are in the right subtree of some node, which means the complexity is also O (log(n)) since we could search along the entire tree.

This function has a pre condition: the node on which the function operates must be real.

The function returns the node with the preceding key in the tree to *this* node's key.

**Helper function:** **public** **void** update(IAVLNode left, IAVLNode right)

O(1) runtime complexity. Change of pointers in constant time.

The function updates the children of the node to be *left* and *right* that we received. Also updates the node as the parent of *left* and *right* nodes.

**AVLTree class:**

This class implements an AVL tree.

Each AVLTree object (a tree) has the following fields:

1. **min -** AVLNode type, which stores the node with the minimum key in the tree.

2. **max -** AVLNode type, which stores the node with the maximum key in the tree.

3. **root -** AVLNode type, which stores the root of the tree.

**Constructor:** **public** AVLTree(AVLNode root, AVLNode min, AVLNode max)

O(1) runtime complexity. Access to fields in constant time.

The function builds an object of type AVLTree according to the given parameters *root*, *min* and *max*, and returns it. If the parameter *root* is a null object, we define the root of the tree to be a virtual node.

**Default constructor: public** AVLTree()

O(1) runtime complexity. Initializing fields in constant time.

The function builds an AVLTree object and returns it, where the object represents an empty tree whose root is a virtual node.

**Function:** **public** **boolean** empty()

O(1) runtime complexity. The function checks whether the root is virtual in constant time.

The function checks whether the tree is empty. The function returns True iff the root of the tree is not a virtual node, i.e. the tree is not empty.

**Function:** **public** String search(**int** k)

O(log(n)) runtime complexity. In the worst case, the loop is executed O(log(n)) times (as the height of the tree) and the operations inside the loop are executed in constant time.

The function searches whether there is a node with the key k in the tree. If exists, the function will return the value of the node. If no such node exists, the function will return null.

The function starts searching starting from the root of the tree. If the key we are looking for is the key of the node we are at, then we have found the desired node and therefore we will return its value. If the key we are looking for is smaller than the key of the node we are at, we will continue the search in the left subtree (using the search tree attribute). Otherwise, the key we are looking for is greater than the key of the current node, so we will continue the search in the right subtree.

**Function:** **public** **int** insert(**int** k, String i)

The function receives an integer *k* representing the key of a node and a string *i* representing the value of the node. The function checks if the key k does not exist in the tree (by calling the treePosition function - details below). If it exists, returns -1. Otherwise, the function builds a new node with the desired key and information and the other suitable fields, and inserts the node in the appropriate place in the tree.

If we inserted the node as a son of a unary node, we update the *size* field of the nodes from the place of insertion up to the root (by calling updateSizesTillRoot - details below). In this case, there is no need to balance, so we return 0 and finish.

Otherwise, we increase the height of the node into which the new node was inserted, and we perform balancing by calling the rebalance function (details below). During the balancing, the heights of the nodes and the sizes of the subtree of each node are updated, the function makes sure that even after the insertion the tree remains a valid AVL tree.

During the balancing, we count the amount of balancing operations performed, and that amount is returned by the function at the end.

The function also takes care of updating the *min* and *max* fields of the tree if necessary, in a constant time, by comparing the current minimum and maximum to the key of the inserted node.

O(log(n)) runtime complexity:

1. Creating a new node and inserting it by updating the pointers: O(1)

2. Calling treePosition: O(log(n))

3. Updating the *min* and *max* fields of the tree: O(1)

4. - In the case of inserting to a unary node: calling updateSizesTillRoot to update the *size* field

of the nodes that need it: O(log(n)).

- in any other case: calling rebalance: O(log(n))

Total in both cases we will get O(log(n)).

**Helper function:** **private** **int** rebalance(IAVLNode z)

The function operates on a tree, it receives a node from which it starts to perform balancing operations to maintain a valid AVL tree. The function lists the problematic cases that arise after an insertion in which the tree is not balanced, according to height differences (5 cases in total, including the symmetrical cases). A repair is made starting from the *z* node towards the root, until we reach a situation where the height differences are valid and maintain the condition of the AVL tree, or until we reach the root.

In order to balance we call helper functions: rotateR, rotateL, promote, demote, diff, and updateSizesTillRoot, all of which work in O(1), except for the last one which works in O(log(n)) but only run once (details below). There is an update of the *size* and *height* fields during the balancing where required.

The function counts the number of balancing operations done and returns this number.

O(log(n)) runtime complexity:

The function traverses nodes from node *z* towards the root, at each node a condition is checked in constant time and the helper functions perform work in constant time. The length of the path where the corrections are made in the worst case is O(log(n)), therefore the total work is O(log(n)). At the end there is a single call to the function updateSizesTillRoot which in the worst case takes O(log(n)), so in total we get O(log(n)).

**Helper function:** **private** **void** rotateR(IAVLNode z, IAVLNode x)

O(1) runtime complexity. Changing pointers in constant time and updating fields in constant time.

The function receives two nodes *z*,*x* and uses them to perform a roll to the right by changing pointers. The function updates the *size* field of the nodes for which this field has changed, which are *x* and *z* nodes.

**Helper function:** **private** **void** rotateL(IAVLNode z, IAVLNode x)

O(1) runtime complexity. Changing pointers in constant time and updating fields in constant time.

The function receives two nodes *z*,*x* and uses them to perform a roll to the left by changing pointers. The function updates the *size* field of the nodes for which this field has changed, which are *x* and *z* nodes.

**Helper function:** **private** **static** **void** promote(IAVLNode z)

O(1) runtime complexity.

The function receives a node *z* and updates its height to be bigger by 1.

**Helper function:** **private** **static** **void** demote(IAVLNode z)

O(1) runtime complexity.

The function receives a node *z* and updates its height to be smaller by 1.

**Helper function:** **private** **static** **int** diff(IAVLNode parent, IAVLNode son)

O(1) runtime complexity. Accesses to fields and an arithmetic operation.

The function receives two nodes, where the first node is the parent of the second node. It calculates the height difference between these nodes.

**Helper function:** **private** **void** updateSizesTillRoot(IAVLNode node)

O(log(n)) runtime complexity. In the worst case, the function will receive as input a leaf and will go through all the nodes in the path from the bottom of the tree to the root. At each node, the size of its subtree will be updated (which is done in constant time). The length of such a path is O(log(n)).

The function receives a node and updates the size of the subtree of the node and of all subsequent nodes in the path from it up to the root.

**Helper function:** **public** AVLNode treePosition(AVLNode node, **int** k)

O(log(n)) runtime complexity. In the worst case, the loop is executed O(log(n)) times (as the height of the tree) and the operations inside the loop are executed in constant time.

The function receives a node and a number *k,* looks in the tree that the received node is its root, searches for a node that supposed to be the parent of a node whose key is k, and returns this node. If there is already a node with key k in the tree, the function will return this node. The function does this using the binary search tree attribute: if the key of the node we are checking is equal to k, then we have found the desired node and therefore we return it. Otherwise, we will check if the key of the node we are at is greater than k, we will continue searching in the left subtree of the node. Otherwise, the key of the node we are at is smaller than k, so we will continue searching in the right subtree of the node.

**Function:** **public** **int** delete(**int** k)

The function deletes from the tree a node with key k if it exists, otherwise it returns -1. A balancing is performed which maintains a valid AVL tree after the deletion. The function counts the number of balancing operations done and returns this number.

To find the node we want to delete, treePosition is called, if it returns a node whose key is different from k then k is not in the tree. The delete function returns -1 and we are done.

Otherwise, if the found node is the only node in the tree, we will turn the tree into an empty tree, that is, we have deleted the node. The function returns 0 because no balancing operations were performed and we are done.

Now there are 3 deletion cases:

* Deleting a leaf: replacing it with a virtual node by calling replaceByVirtual (details below).
* Deleting a unary node: if it is the root, we find which of its children is real, and make it the new root of the tree. So essentially the original root is deleted. Otherwise, we delete the node by bridging over it between his father and his real son, this by calling bypass (details below).
* Deleting a binary node: we find its successor by calling successor(), replace between them by replace (details below), and then delete the successor which must be a leaf or a unary node as described in the above cases.

After the deletion, the function begins to perform the balancing operations, according to the problematic cases as well as the symmetric cases (7 cases in total). The balancing starts from the parent of the deleted node towards the root, until the problem is solved (the height differences are legal and maintain the condition of the AVL tree, or we have reached the root).

In order to balance we call helper functions: rotateR, rotateL, promote, demote, diff, all of which operate in O(1) as detailed above. There is an update of the *size* and *height* fields during the balancing where required.

At the end, we check if we have traversed all nodes in the path from the starting point of the balancing up to the root. If not - we update the *size* field of the remaining nodes up to the root by calling the updateSizesTillRoot function. In the worst case no balancing will be performed at all and the function has to go through the entire path to the root, and such a path will cost O(log(n)).

There is an update of the *min* and *max* fields of the tree if necessary, that is, if the minimum is deleted, we would like to find the new minimum by finding its successor (calling successor()), and if the maximum is deleted, then we will find the new maximum by finding its predecessor (calling predecessor()).

At the end, the function returns the number of balancing operations that were performed.

O(log(n)) runtime complexity:

1. Calling treePosition: O(log(n)).
2. Deleting the node: if a leaf or a unary node is deleted then replaceByVirtual and bypass are called respectively: O(1), and in the case where a binary node is deleted, we call successor() which costs O(log(n)) and replace which costs O(1), and then we delete like a leaf or a unary node: O(1). i.e., in the worst case, deletion costs O(log(n)).
3. Balancing: O(log(n)) if we had to go up to the root and only there the problem was solved (at each node performed operations in O(1)).
4. Calling updateSizesTillRoot to update the *size* field of the nodes that need it: in the worst case O(log(n)).
5. Updating the *min* and *max* fields of the tree: in each deletion we would have to update at most one of them, that is, there will be at most one call to successor() or to predecessor() which costs O(log(n)).

The operations are performed sequentially and not in parallel, so we add them up and get O(log(n)).

**Helper function:** **private** **void** replace(AVLNode x, AVLNode s)

O(1) runtime complexity. Accessing and updating fields.

The function receives two real nodes, where the second is the successor of the first, and switches between their keys and their values. Thus we will get the info and key of *s* at the location of *x*, and vice versa. This function is required as part of deleting a node with two children (a binary node).

**Helper function:** **private** **void** bypass(AVLNode y)

O(1) runtime complexity. Accessing fields, updating fields, and changing pointers.

The function receives a unary node *y*, finds which of its children is a real child, and connects *y*'s parent to its real child by changing pointers. This way, *y* is essentially deleted from the tree.

**Helper function:** **private** **void** replaceByVirtual(AVLNode x)

O(1) runtime complexity. Calling the default constructor, accessing fields, updating fields, and changing pointers.

The function receives a node *x* which is a leaf, and replaces it with a virtual node (while updating the relevant pointers). This way, the leaf is essentially deleted from the tree.

**Function:** **public** String min()

O(1) runtime complexity. Access to the value of the field *min* in constant time.

The function returns the value of the node with the minimum key, or null if the tree is empty.

**Function:** **public** String max()

O(1) runtime complexity. Access to the value of the field *max* in constant time.

The function returns the value of the node with the maximum key, or null if the tree is empty.

**Function:** **public** **int**[] keysToArray()

O(n) runtime complexity. The function performs actions whose time is constant but it also calls the function **public** **int** keysToArray(AVLNode node, **int**[] arr, **int** i) whose complexity is O(n), therefore the total complexity is O(n).

The function returns a sorted array (from smallest to largest) containing the keys of the tree. First the function defines an array of integers in the size of the tree. Then it calls the function **public** **int** keysToArray(AVLNode node, **int**[] arr, **int** i), and at the end returns the array.

**Function:** **public** **int** keysToArray(AVLNode node, **int**[] arr, **int** i)

O(n) runtime complexity. This function traverses all the nodes in the tree in an in-order walk to put their keys in a sorted order in an array.

A recursive function that receives an array of integers *arr*, an integer *i* that represents the index in the array to which the next element will enter, and the node *node*. The function fills the array *arr* with keys in ascending order, this way the array is sorted at the end of the function run. The function returns the next index at the array to which the next element should be inserted.

**Function:** **public** String[] infoToArray()

O(n) runtime complexity. The function performs actions whose time is constant but it also calls the function **public** **int** infoToArray(AVLNode node, String[] arr, **int** i) whose complexity is O(n), therefore the total complexity is O(n).

The function returns an array of strings containing the values ​​of the nodes of the tree in sorted order (according to the keys of the tree, from the smallest to the largest). First, the function defines an array of strings in the size of the tree. Then, it calls the function

**public** **int** infoToArray(AVLNode node, String[] arr, **int** i), and at the end returns the array.

**Function:** **public** **int** infoToArray(AVLNode node,String[] arr, **int** i)

O(n) runtime complexity. This function traverses all the nodes in the tree in an in-order walk to fill the array in a sorted order.

A recursive function that receives an array of strings *arr*, an integer *i* that represents the index to which the next element in the array will enter, and the node *node*. The function fills the array *arr* with the values ​​of the tree nodes in ascending order according to their keys. The function returns the next index at the array to which the next element should be inserted.

**Function:** **public** **int** size()

O(1) runtime complexity. Accessing the field *root* and calling the getSize function. These actions take constant time.

The function returns the size of the tree, i.e. the number of nodes in it.

**Function:** **public** IAVLNode getRoot()

O(1) runtime complexity. If the tree is not empty, the field *root* is accessed, which takes constant time.

The function returns the root of the tree if the tree is not empty, otherwise it returns null.

**Function:** **public** **int** join(IAVLNode x, AVLTree t)

The function operates on a tree, receives a node *x* and another tree *t*, where all the keys in one of the trees are smaller than *x*'s key, and all the keys in the other tree are larger than *x*'s key. It combines them and then balances to get a valid AVL tree.

We will refer to the tree on which the function operates as t1, and the tree the function receives as t2, for convenience.

The function performs joining according to the specific case, which can be one of the following:

1. t1 is empty and t2 is not empty - we insert *x* into t2 and update all t1's fields to be t2's fields (including the root).
2. Both trees are empty - we update the root of t1 and its *min* and *max* fields to be *x*.
3. t1 is not empty and t2 is empty - we insert *x* into t1.
4. Both trees are not empty and at the same height: we update *x* to be the root of t1 and its children to be the roots of t1 and t2 (in the appropriate order according to the roots' keys). No need for balancing.
5. Both trees are not empty and at different heights: we check which tree is higher. We refer the root of the lower tree as *a*.

* If the keys of the higher tree are greater than *x*, we go from its root down on the leftmost branch until we find a node whose height is as *a*'s height or less than one, we refer it as *b* (by the function findPlaceLeft - details below). We set *a* and *b* as *x*'s children by the update function, and set *x*'s parent to be *b*'s parent.
* If the keys of the higher tree are smaller than *x*, we go from its root down on the rightmost branch until we find a node whose height is as *a*'s height or less than one, we refer it as *b* (by the function findPlaceRight - details below). We set *a* and *b* as *x*'s children by the update function, and set *x*'s parent to be *b*'s parent.

After the connection, we call rebalanceForJoin (described below) which balances the tree after the join operation.

Throughout the process, care is taken to update the *min* and *max* fields of t1 and the *size* and *height* fields of the nodes by constant time operations (O(1)).

The function returns the cost of the join operation (height difference between the trees + 1).

O(log(n)) runtime complexity: we will list the complexity of each case:

1. Cost of insert + constant time operations, in total: O(log(n)).
2. Updating pointers and fields: O(1).
3. Cost of insert: O(log(n)).
4. Updating pointers and fields and calling update: O(1).
5. Calling findPlaceLeft or findPlaceRight : O(log(n))

Calling update: O(1)

Changing pointers, *height* and *size* fields: O(1)

Calling rebalanceForJoin : O(log(n))

Total : O(log(n))

In any case, the complexity does not exceed O(log(n)).

**Helper function:** **private** **void** rebalanceForJoin(AVLNode z, AVLNode y)

The function performs balancing after a join operation has occurred. The node *z* is the parent of *x* (the node that join received), and the node *y* is the parent of *z*.

There are two problematic cases (symmetrical) after a join operation in which the invariance of an AVL tree is not preserved, and are not included in the problematic cases after insertion. Therefore, the function checks these cases. If one of the cases holds, the function repairs, and sends *y* to the rebalance function that is used by insert (i.e. the balancing should only start from *z*'s parent because we have already made repairs for *z*). Otherwise, the function sends *z* to the rebalance function (start balancing starting from *z*).

O(log(n)) runtime complexity:

If one of the cases in the function holds then balancing operations occur in constant time O(1).

There is a call to rebalance that costs O(log(n)). Therefore, in total we get O(log(n)).

**Helper function:** **private** **static** AVLNode findPlaceLeft(AVLTree tree, **int** k)

O(log(n)) runtime complexity. In the worst case, the loop is executed O(log(n)) times (as the height of the tree) and the operations inside the loop are executed in constant time.

The function receives a tree and a number *k* and returns a node from the left branch of the tree, so that its height is equal to *k* or *k*-1 (depending on what it found first). The function starts from the root of the tree and continues to go over the left branch if we have not yet reached the appropriate node.

**Helper function:** **private** **static** AVLNode findPlaceRight(AVLTree tree, **int** k)

O(log(n)) runtime complexity. In the worst case, the loop is executed O(log(n)) times (as the height of the tree) and the operations inside the loop are executed in constant time.

The function receives a tree and a number *k* and returns a node from the right branch of the tree, so that its height is equal to *k* or *k*-1 (depending on what it found first). The function starts from the root of the tree and continues to go over the right branch if we have not yet reached the appropriate node.

**Function:** **public** AVLTree[] split(**int** x)

The function splits a tree into two trees according to *x*'s key - a tree that contains all keys smaller than *x*'s and a tree that contains all keys greater than *x*'s.

A pre condition for the function: the key *x* is in the tree.

The function returns an array of two trees, where the first tree is the small keys tree, and the second is the large keys tree.

The function searches for the node with the key *x* by calling treePosition.

It creates two trees – 'smaller' which is a tree rooted by the left child of the node we found, and 'bigger' which is a tree rooted by the right child of the node we found.

If the node we found is the root of the tree the function operates on, then we update the *min* and *max* fields of both new trees and return the array {'smaller','bigger'} (i.e., the initial definition of these trees is already the needed split).

Otherwise, we disconnect the node we found from its parent *z*, and decide if *z*'s key is greater than *x*'s. If so - we perform a join operation on the tree 'bigger', with the node *z* and a new tree rooted by the right child of *z*. If not - we perform a join operation on the tree 'smaller', with the node *z* and a new tree rooted by the left child of *z*.

We continue the same process for z's parent, until we reach the root's parent (which is null). That is, the length of the route is at most O(log(n)).

At the end, the *min* and *max* fields of the trees 'smaller' and 'bigger' are updated by calling the min() and max() functions (if necessary). The array {'smaller','bigger'} is returned.

O(log(n)) runtime complexity.

1. Calling treePosition: O(log(n)).
2. Creating the two trees (calling the constructor of a tree): O(1).
3. If the node we found is the root of the tree - we are done.

Otherwise, we go from the node whose key is *x* up to the root and perform an efficient join. The complexity of this process is O(log(n)).

1. Calling min() and max(): O(log(n))

The callings are performed sequentially and not in parallel, therefore we will add up and get a total of: O(log(n)).