**תרגיל מעשי 1 – מבני נתונים**

ניתוח סיבוכיות זמן הריצה:

AVLTree methods:

**Built-in Methods:**

* **public boolean empty() –** the method checks if the root is "null", or if it's not a real-node. Therefore, complexity = O(1).
* **public String search(int k) –** the method search the info of a node, and uses the method – "search\_recursive" which has a complexity of O(log(n)) (will be elaborated). Therefore, search method has a complexity of O(log(n)).
* **public int insert(int k, String i) –** the method inserts an item with key "k" and info "i". the method firsts check if the tree is empty, if so it adds the node to the empty tree – complexity = O(1). Otherwise, if the key "k" is already in the tree (call "search" method), it returns "-1", complexity = O(log(n)) of "search".

Otherwise, the method create a new node and insert it to the tree using "regular\_insert" method (will be elaborated), and balance the AVL tree using "balance2\_for\_insert\_and\_join" method (will be elaborated). Both methods we used costs O(log(n)), therefore the complexity in the worst-case for insert is O(log(n)).

* **public int delete(int k) –** the method deletes an item with key k.

first, the method checks if the given key is in the tree, if not – it returns -1. The complexity is O(log(n)), using search method.

If the key is found, the method calls "delete\_regular" method (will be elaborated) which has a complexity of O(log(n)). Then, the method checks if the AVL tree is now empty – if so it updates its values to null and return 0. The method then balances the tree using the method "balance1\_for\_delete" (will be elaborated) which has a complexity of O(log(n)). Afterwards it deals with the cases in which the "minimum"/"maximum" values were deleted and set the new values. Both cases have O(log(n)) complexity due to the height of the AVL tree.

* **public String min() –** the method returns the info of the item with the smallest key. Our method firsts checks if the tree is empty – if so it returns null. Otherwise it returns the value "minimum". Both cases have a complexity of O(1).
* **public String max() –**  the method returns the info of the item with the biggest key. Our method firsts checks if the tree is empty – if so it returns null. Otherwise it returns the value "maximum". Both cases have a complexity of O(1).
* **public int[] keysToArray() –** the method returns a sorted array containing all keys. The method first create an array containing the number as the tree's size. Then, it finds the minimum value in the tree using the value "minimum". Later on, using the "Successor" method (will be elaborated)adds the items to the created array**.** We saw in the recitation that using n-times successor when having a pointer to the minimum-value – costs O(n), like going in-order. Therefore, a total of O(n) complexity.
* **public String[] infoToArray() –** same as "keysToArray()" only sorting the values instead of the keys. Therefore O(n) complexity.
* **public int size() –** the method returns the number of nodes in the tree. If the tree is empty it returns 0, otherwise the root's value – size. Therefore O(1) complexity.
* **public int getRoot() –** the method returns the tree's root. Therefore uses the value "root". So, O(1) complexity.
* **public AVLTree[] split(int x) –** the method splits the tree into 2 trees. The method first creates new AVL trees – "big, small", O(1) complexity. Then the method checks if the "connector" node is a "right child" or "left child" using its keys. The checking is also O(1) complexity. Then the method uses "join" method for the chosen case (will be elaborated) which has a O(log(n)) complexity**.** Afterwards we adjust the values of the new trees and return them in an array. Therefore, total complexiy of O(log(n)).
* **public int join(IAVLNode x, AVLTree t) –** themethod joins the given node and tree to the current tree.

First, the method checks if both trees are empty – if so it inserts the given node to the current tree and returns 1 – O(1) complexity.

Otherwise, if the current tree is empty or the given tree is empty it inserts the node to the non-empty tree and return it's height + 1.

Both of these cases uses "insert" method which has a complexity of O(log(n)).

If the above cases weren’t relevant, the method calculates the rank difference between the two trees, to be returned at the end. Then, according to the chosen cases among the follows, the method adjusts the tree:

1. the current tree is shorter than the given tree – using "case\_shorter" method" (will be elaborated).
2. The current tree is at the same height and also connector's – we use "case\_equal" method (will be elaborated).
3. Else, the current tree is higher than the given, so we call "case\_higher" method.

All of the cases update and set the tree to the given situation, and also checks the exact location to "join", therefore have O(log(n)) worst-case complexity. At the end we rebalance the tree of needed with the method "balance2\_for\_insert\_and\_join" which costs O(log(n)) as said. At last, we return the rank difference as said. Total complexity of O(log(n)).

**Added Methods:**

* **private void adjust\_max() –** the method adjusts the tree's "maximum" value, to it's current max. we use the AVL quality in which the maximum key is the most-right, and so we travel until reaching the right edge. Therefore a complexity of O(log(n)).
* **private void adjust\_min() –** the method adjusts the tree's "minimum" value, to it's current min. we use the AVL quality in which the minimum key is the most-left, and so we travel until reaching the left edge. Therefore a complexity of O(log(n)).
* **private IAVLNode search\_this\_node(int p) –** the method supports the "search" method. It returns the node who's key was given. It uses the method "rec\_srch" (will be elaborated), therefore has a complexity of O(log(n)).
* **private String search\_recursive\_for\_val (int p, IAVLNode node) –** the method supports the "search" method. The current method uses the binary-tree qualities in order to look for the given key in the tree in a recursive way. We know that binary-search "costs" O(log(n)) in AVL due to the height of the tree, so the complexity is O(log(n)).
* **private IAVLNode search\_recursive\_for\_node(int p, IAVLNode node) –** the method supports the "search" method. It works like "search\_recursive" method, only here we return the node itself and not its value. Therefore, as said before – a complexity of O(log(n)).
* **private void switchthis(IAVLNode first, IAVLNode second) –** the method supports the "delete" method. It switches positions of given nodes. The method only sets/adjusts values, so total complexity of O(1).
* **private int balance1\_for\_delete(IAVLNode n) –** the method commits the "balance" action as learned, and supports the "delete" method in particular. Along the method we verify that the balance is set (we operate in a while-loop until it is) by checking the height difference between the two children. If needed we call the methods "rotate\_to\_the\_left/right" which costs each a O(1) (will be elaborated). Therefore by using these methods during the while-loop, we know that the "balancing"action can be done up until the root, which is up to a log(n) height (due to AVL qualities). Therefore it's a complexity of a constant times a log, which means complexity of O(log(n)).
* **private int balance2\_for\_insert\_and\_join(IAVLNode n) –** as we explained in the previous method, this method commits the "balance" action and supports the "insert" and the "join" methods. The function commits rebalancing using the methods "rotate\_to\_the\_left/right" which costs each O(1) (will be elaborated). Therefore, as said, we commit these rebalancing operations up until the root (worst-case), which is at log(n) height due to AVL qualities. Then, as said previously we have a complexity of O(log(n)).
* **private IAVLNode deleteregular(IAVLNode to\_delete) –** the function commits the regular deletion as in binary search tree and returns the node in which the rebalancing process is from. Worst-case scenario in this method is when the node "to\_delete" have both right and left child and we are looking for a successor. In this scenario we can go down until the leaves and the bottom of the tree. As said before the height of AVL tree is logarithmic, therefore we have a complexity of O(log(n)).
* **private void regular\_insert(IAVLNode z) –** this method supports the "insert" method, and commit an insert operation as in binary search tree. The key to be inserted can be at the bottom of the tree, and the insertion process as we know, starts from the top. So, in worst case, we can go down until the bottom of the tree, and as said the height of AVL is logarithmic, so we have a complexity of O(log(n)).
* **private IAVLNode rotate\_to\_the\_left(IAVLNode rotated) –** this explanation is also correct for the next method which is "rotate\_to\_the\_right". This method commits the rotation action as learned in class. The function commits the rotation and returns the root after rotation. Along the method we have no loops/using other method, therefore we only adjusting values and replacing nodes. So, a total complexity of O(1).
* **private IAVLNode rotate\_to\_the\_right(IAVLNode rotated) –** as explained above.
* **private static IAVLNode Successor(IAVLNode node) –** this method commit the "successor" operation as learned in class. If the node has a right child it returns it, and otherwise returns the node's successor. The successor can be (in worst-case) at a distance of "tree-height" from the node, and as said the height of AVL tree is logarithmic. Therefore this method complexity is O(log(n)).
* **private void case\_equal(AVLTree tree\_first, AVLTree tree\_second, IAVLNode connector) –** this method supports the "join" method. In this method we take care of the case in which the two trees are at the same level. In the method we set the new children to the "connector" node and adjust the tree's values. Therefore, a total complexity of O(1).
* **private void case\_shorter(AVLTree tree, IAVLNode node\_1) -** this explanation is also correct for the next method which is "case\_higher". This method also supports the "join" method, and takes care of the case in which current is shorter then the second tree. Therefore we are looking for the node to join them. This search can be down until the bottom of the tree, which means it bounded by log(n) (AVL height). Therefore it's a total complexity of O(log(n)).
* **private void case\_higher(AVLTree tree, IAVLNode node\_1) –** as explained above.

IAVLNode methods

**Built-in methods:**

* **public int getKey() –** returns the value "key". Therefore O(1).
* **public String getValue() –** returns the value "info". Therefore O(1).
* **public void setLeft(IAVLNode node) –** sets current's "left" to node. Therefore O(1).
* **public IAVLNode getLeft() –** returns current's "left". Therefore O(1).
* **public void setRight(IAVLNode node) -** sets current's "right" to node. Therefore O(1).
* **public IAVLNode getRight() –** returns current's "left". Therefore O(1).
* **public void setParent(IAVLNode node) –** sets current's "parent" to node. Therefore O(1).
* **public IAVLNode getParent() –** returns current's "parent". Therefore O(1).
* **public boolean isRealNode() –** checks if key is not "-1". Therefore O(1).
* **public void setHeight(int height) –** set's current's "height" to the given value. Therefore O(1).
* **public int getHeight() –** returns current's "height". Therefore O(1).

**Added methods:**

* **public void adjustheight() –** adjusts curren'ts height by his children's height. Therefore O(1)
* **public void adjust\_size() –** adjusts current's size by his children's sizes. Therefore O(1).
* **public int get\_size() –** returns current's size. Therefore O(1).
* **public void adjust\_balance() –** adjusts current's balance by his children's balances. Therefore O(1).
* **public int getbalance() –** returns current's balance. Therefore O(1).
* **public void virtualize() –** creates two virtual nodes and assign them as children to current. Therefore O(1).
* **public boolean check\_my\_balance() –** check if current's balance is ok (due to AVL rules). Therefore O(1).