**Assignment 1 – Data Structures**

**Class 1 – AVLNode**

**Description –** A class representing a single node in the list. The node consists of several fields:

|  |  |  |
| --- | --- | --- |
| Field | Description | Type |
| Value | Each node stores a certain value | Any |
| Key | Each node has a key that determines its position in the list | None/int |
| Left | Reference to the left child of the node | AVLNode/None |
| Right | Reference to the right child of the node | AVLNode/None |
| Parent | Reference to the parent of the node | AVLNode/None |
| Height | An integer representing the length of the longest path down the tree from the current node to a leaf. If the node is virtual the height will be -1 | Int |
| Size | An integer representing the amount of nodes in the subtree from the node (including the node itself) | int |

The class supports the regular set/get methods in an object and also supports the methods below.

|  |  |  |
| --- | --- | --- |
| Method | Description | Complexity |
| get\_key(self) | Returns from the given node the key, None if self is virtual |  |
| get\_value(self) | Returns from the given node the value, None if self is virtual |  |
| get\_left(self) | Returns a reference to the left child from the given node, None if self is virtual |  |
| get\_right(self) | Returns a reference to the right child from the given node, None if self is virtual |  |
| get\_parent(self) | Returns a reference to the parent of the given node. |  |
| get\_height(self) | Returns an integer representing the height of the given node as described before. |  |
| get\_size(self) | Returns an integer representing the size of the given node as described before |  |
| set\_key(self,key) | Sets/Overwrites to the given node the given key |  |
| set\_value(self,key) | Sets/Overwrites to the given node the given value |  |
| set\_left(self,node) | Sets/Overwrites to the given node the reference to the left child |  |
| set\_right(self,node) | Sets/Overwrites to the given node the reference to the right child |  |
| set\_parent(self,node) | Sets/Overwrites to the given node the reference to the parent |  |
| set\_size(self, size) | Sets/Overwrites the size of the given node with the given size |  |
| is\_leaf(self) | Returns true if the right and left children of the given node are virtual nodes, false otherwise |  |
| is\_real\_node(self) | Returns true if the given node is virtual, false otherwise |  |
| min\_node(self) | Returns the node with the minimum key from the given node by traversing left until reaching a node without a left child (the returned node has a virtual node in it’s left reference) | The method traverses left from the given node using a while loop, the number of iterations is at most the height of the tree. And we have proved in class that h = O(log(n)), when n represents the number of elements currently in the tree. |
| in\_order\_successor(self) | Returns the successor node in an in-order method from the given node, by calling the min\_node method from the right child, if the right child is not virtual, otherwise returns the given node itself. | O(log(n))  The cost of checking if the right child exists and returning it is calling the min\_node method is it’s main cost and as mentioned before it is |
| updateHeight(self) | Overwrites the given node’s height. If the node is a leaf, we update its height to 0. Otherwise, we update its height to be max of the heights of it’s right and left children plus one. | The cost of the methods that are being called and used, max, get and set is constant. |
| updateSize(self) | Overwrites the given node’s size, by setting it to the sum of the right and left children sizes plus 1. | The cost of set that is being called and used several times is constant. |

**Class 2 – AVLTree**

**Description:**

A list that is represented by an AVL tree that consists of AVLNode objects. Each tree has size, root.

The class supports get and set for the size, root fields, and the methods below

|  |  |  |
| --- | --- | --- |
| Method | Description | Complexity |
| search(self,key) | The method gets a key and searches using an inner recursive method to search a node with the given key. | O(n)  The worst case is when the method is given a key that doesn’t exist, the method will go over all elements in the tree resulting in going over n elements. |
| search\_rec(self,key) | The helper function for the method mentioned above | O(n) as we’ve mentioned above |
| insert(self,key,val) | This function works the same way as we’ve learned in class, besides maintaining height and size which is constant. the main difference is that while we iterate from the inserted node back to the root of the tree, if we’ve didn’t encounter a change in the height we still don’t break since we need to update the size up to the root | O(log(n)), this method calls a regular insert to binary search tree method which has complexity of O(log(n)), after that we iterate from the given node up to the root, since we have proved that the height of an AVLTree is O(log(n)), we get in total O(log(n)) the recursive method and the loop are independent. |
| insert\_rec(self,node,key,val) | This method is a regular insertion to a binary search tree | O(log(n))  This method is known to have this complexity |
| BFS(self, node) | This method calculates the balance factor of a given node in the tree | This method uses an arithmetic expression which is constant |
| rotate(self,node,BFS) | Given a node and its balance factor, the method checks what kind of rotations are needed and executes them with the methods accordingly | This method uses conditions and basic expressions without using any loops or recursion. |
| left\_rotate(self,node) | Given a node that needs to be rotated to the left the method changes the references accordingly to what we learned in class | This method uses basic expressions using reference calls which are constant. |
| right\_rotate(self,node) | The same as the previously mentioned method but to the opposite direction |  |
| left\_then\_right(self,node) | This method executes two rotations using the methods mentioned above | The methods that being called are constant resulting in total constant time |
| right\_then\_left(self,node) | Works the same as mentioned above |  |
| fix\_parent(self,A,B) | Given two nodes after rotation, this method overwrites the references to the parent to match the right references | This method uses basic expressions using references without any loops or recursion resulting in a constant time |
| delete(self,node) | Given a node reference, this method works the same way as we learned in class, after calling the BFS\_delete method which uses basic deletion algorithm in Binary Search Tree, we iterate to the root from the parent of the physically deleted node making sure the tree is indeed an AVLTree | O(log(n))  This method uses BFS\_delete method which is similar to the regular deletion from a Binary Search Tree algorithm with small modifications which are constant, resulting in the mentioned complexity |
| BFS\_delete(self, node, key) | Given the root of the tree, we iterate with recursion to reach the node that needs to be deleted, this method uses in\_order\_successor in order to get the successor node that will replace the node with the given key. | O(log(n))  This method works similarly to the regular deletion in BST, the main differences are that after reaching the node we change parent reference. After using recursion to reach the node with the given key, with complexity of O(log(n)), we use independently in\_order\_successor method with complexity of O(log(n)) resulting in overall mentioned complexity |
| avl\_to\_array(self) | This method converts the tree into an array using an inner recursive helper function | Since we iterate over all the elements in the tree every time the method is being called, we get the mentioned time complexity |
| avl\_to\_array\_rec(array,node) | This is the inner recursive function to the method mentioned above |  |
| size(self) | This method returns the size of the root of self. | Constant time calling the field |
| split(self) | This method splits the tree to two AVLTrees given a node. This algorithm works as we learned in class | We have proved in class that this method works in the mentioned complexity |
| join(self,tree,key,val) |  |  |
| Balance(self,node) |  |  |
| rank(self,node) | This method calculates the rank of a given reference to a node in self. This function uses the algorithm based on what we learned in class | We have proved in class that this method works in the mentioned complexity |
| select(self,node,i) | This method finds the i’th smallest element (according to keys) in self | We have proved in class that this method works in the mentioned complexity |
| get\_root(self) | This method returns a reference to the root of self | Returning a reference in constant time |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Cost of AVL sorting for almost sorted array | Number of transpositions in almost sorted array | Cost of AVL sorting for shuffled array | Number of transpositions in shuffled array | Cost of AVL sorting for reversed-sorted array | Number of transpositions in reversed-sorted array | Index of i |
| 43107 | 451500 | 56557 | 2272816 | 58912 | 4501500 | 1 |
| 92189 | 903000 | 120942 | 9014150 | 129757 | 18003000 | 2 |
| 190356 | 1806000 | 272993 | 36182682 | 283435 | 72006000 | 3 |
| 386651 | 3612000 | 608219 | 142629193 | 614778 | 288012000 | 4 |
| 779227 | 7224000 | 1299511 | 576908146 | 1325450 | 1152024000 | 5 |

To calculate efficiently the number of transpositions, we have used the method rank on each node that was inserted, with the expression key – rank + 1. Since the expression key – rank represents the number of keys that are missing as we established in theoretical assignment 2.

1. . The cost of searching the i’th element in the tree is

The values in the table in comparison to the analysis is valid, since the table values are less than twice as big, so asymptotically we have correlation.