**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) | This method traverses to the minimum node in the subtree of node, using a while loop with getting the left son each iteration until we reach a leaf which happens to be the minimum node. | Since we traverse left each iteration until reaching a leaf, the maximal distance from a leaf from the given node is bounded by the height of the tree which is bounded by the desired complexity. |
| 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 its 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 root and a reference to its minimum node. The size of the root is determined by the node that is saved as root, if no node is saved as the root meaning the root is none, we return 0.

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 helper method will go over all elements in the tree resulting in visiting all n elements. |
| search\_rec(node,key) | This method searches the subtree with node as the root recursively, going left when the current node’s key is smaller than the given key, and right otherwise. We return None if we didn’t manage to find the node with the given key. | O(n) as we’ve mentioned above |
| insert(self,key,val) | This function works the same way as we’ve learned in class, we first use a recursive insertion method to insert a node, after finishing inserting we take the parent of the inserted node, and start going up the tree using call for parent each iteration using a while loop. While we iterate we calculate the balance factor using a helper method, and update size/height as we go up the tree. | O(log(n)), this method uses the helper method insert\_rec which as explained below with complexity of O(log(n)), after inserting the node we use a while loop from the parent of the inserted node going up the tree until we reach the node. The number of iterations is bounded by the tree’s height, and as we established it is also bounded by the desired complexity. Because the insertion is called outside the while loop we get overall the desired complexity. |
| insert\_rec(self,node,key,val) | This method uses recursive search in order to reach the insertion placement, meaning we keep going down the tree recursively until we reach a leaf, each time we visit a node we check if we need to go left or right by comparing the given key with the current node’s key. | O(log(n))  This method goes down the tree until we reach a leaf, the maximal distance between the root and a leaf is the height of the tree, as we established in class the height is bounded, resulting in the desired 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 | Same as explained above. |
| 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 | Same as explained 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, we firstly call BFS\_delete in order to physically delete a node, and then we get the parent of the physically deleted node and iterate up the tree using a while loop as we check each iteration the balance factor using the BFS method, and updating the size/height of each node we visit in order, we also occasionally call the rotate method if we reach an unbalanced node. | O(log(n))  This method uses BFS\_delete which works in O(log(n)) complexity as mentioned below, after deleting we use a while loop iterating from the parent of the physically deleted node up the tree, the number of iterations is bounded by the height of the tree, which is also bounded by the desired complexity. As a result of using the while loop after the deletion we get overall the desired complexity. |
| BFS\_delete(self, node, key) | This function receives the root of the tree, after that we start searching the node we want to delete recursively, after reaching the desired node we get the in-order successor of the node using the in-order-successor method in order to replace the node with the successor data and then physically deleting the in-order successor. | O(log(n))  This method uses recursive search in order to get to the desired node, with time complexity of O(log(n)), after that we get the in-order successor using the method in-order-successor, which also has time complexity of O(log(n)), as mentioned before, because we use call the in-order-successor method independently from searching, we get a total time complexity of O(log(n)) |
| 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 method uses two recursive calls that result in visiting all the nodes in the tree, between the recursive calls we insert each node to the array, resulting in inserting the nodes in an in-order method. | Explained above |
| size(self) | This method returns the size of the root of self. | Constant time returning the desired field. |
| split(self) | This method splits the tree to two AVLTrees given a node. This algorithm works as we learned in class | As we have established in class the overall complexity is O(log(n)) |
| join(self,tree,key,val) |  |  |
| Balance(self,node) |  |  |
| rank(self,node) | This method calculates the rank of a given node, using the algorithm we saw in class. We first calculate the size of the left subtree + 1 (including the node itself). And then we iterate up the tree as long the parent is from the left accumulating the size of the left subtree plus one. If we reach a parent from the right inside the loop or reach the root of the tree we stop and return the accumulated variable. | Since we traverse up from the given node up the tree, the total number of iterations is bounded by the height of the tree, which is bounded by the desired complexity. |
| select(self,node,i) | This method uses a while loop from the minimal node to traverse up the tree until we reach a node with a size of at least i, then it uses helper function that uses recursion to find the i’th smallest element (according to keys) in self. | Since the method uses while loop to reach a node with at least size i, the cost of reaching that node is log(i) as we have proved in the theoretical assignment 2. |
| select\_min(node, i) | This method uses recursion traversing down the tree from the given node until we reach a node that has it’s left size + 1 equal to i. | O(log(i))  The recursion starts from the node with size of at least i, traversing down the subtree until reaching the desired node, the number of recursion calls is bounded by the height of the subtree, which is O(log(i)), resulting in the desired complexity. |
| get\_root(self) | This method returns a reference to the root of self | Constant time returning a reference |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Cost of search for almost sorted array | Number of transpositions in almost sorted array | Cost of search for shuffled array | Number of transpositions in shuffled array | Cost of search for reversed-sorted array | Number of transpositions in reversed-sorted array | Index of i |
| 42602 | 451500 | 51676 | 2272816 | 58836 | 4501500 | 1 |
| 90762 | 903000 | 117888 | 9014150 | 129668 | 18003000 | 2 |
| 187082 | 1806000 | 257696 | 36182682 | 283332 | 72006000 | 3 |
| 379722 | 3612000 | 558461 | 142629193 | 614660 | 288012000 | 4 |
| 765002 | 7224000 | 1240817 | 576908146 | 1325316 | 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 , since the table values are almost twice as big, so asymptotically we have correlation.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Index | Average cost of join for random split | Maximal cost of join for random split | Average cost of join for splitting the maximal node in the left sub tree | Maximal cost of join for splitting the maximal node in the left sub tree |
| 1 | 1.595 | 7 | 1.040 | 14 |
| 2 | 1.710 | 7 | 1.039 | 15 |
| 3 | 1.615 | 6 | 1.028 | 16 |
| 4 | 1.683 | 6 | 1.006 | 17 |
| 5 | 1.662 | 7 | 1.013 | 18 |
| 6 | 1.729 | 6 | 1.016 | 20 |
| 7 | 1.749 | 5 | 1.018 | 21 |
| 8 | 1.747 | 8 | 1.008 | 22 |
| 9 | 1.755 | 5 | 1.037 | 23 |
| 10 | 1.739 | 7 | 1.016 | 24 |